The Elastic Certified Engineer Exam v8.1 Study Guide

Table of Tasks

Welcome

Welcome to the official website for **The Definitive Guide to the Elastic Certified Engineer Exam v8.1**. This resource aims to cover the essential topics you need to know for the Elastic Certified Engineer Exam, providing practical examples and insights to help you succeed.

As you prepare for the exam, you'll also discover additional aspects of Elastic that might surprise you. Embrace the learning process, as every bit of knowledge gained will enhance your skills as an Elasticsearch developer.

This website is hosted on GitHub and will always be free. However, please note that the Elastic exam will eventually update from Elasticsearch v8.1 to newer versions as they become available.

If you encounter any issues or have suggestions, please report them here. I strive to update the content promptly and appreciate your patience!

Please remember that the content on this website is protected and may not be copied or reproduced without permission. The code examples are concise and meant to be copied directly from the HTML pages into the Elastic Kibana Console for easy use.

If you find this online guide helpful, consider supporting us by purchasing a paper copy or the eBook from your favorite vendor.

Happy studying, and best of luck on your journey to becoming an Elastic Certified Engineer!

Preface

Welcome to the **Elastic Certified Engineer Exam v8.1 Study Guide**! This guide is designed to help you navigate the Elastic Certified Engineer exam, as detailed on the official certification exam landing page. Our goal is to provide you with the knowledge and tools you need to not only prepare for but also pass the exam on your first attempt.

This guide is packed with numerous examples covering the various topic areas outlined in the exam. The focus is on understanding Elasticsearch scenarios and exploring potential solutions to the challenges you might encounter. By working through these examples, you will gain practical insights that are crucial for the exam and for your development as an Elasticsearch professional.

For your convenience, this book is also available online for easy browsing at **Elastic Certified Engineer Exam v8.1 Study Guide**.

Acknowledgments

Creating this guide involved many weeks of dedicated effort and collaboration with numerous Large Language Models (LLMs). I pushed and prodded these models to generate a wide range of examples that comprehensively cover the areas described by Elastic. The absence of a comprehensive guide inspired me to create this resource, and I sincerely hope it proves as beneficial to you as it has to me.

Thank you for choosing this guide, and I wish you the best of luck on your journey to becoming an Elastic Certified Engineer!

Introduction

This study guide will go over the specific study areas as listed by Elastic on their website for the Elastic Certified Engineer Exam v8.1 (true as of XXX NN, 2024):

Data Management

- Define an index that satisfies a given set of requirements
- Define and use an index template for a given pattern that satisfies a given set of requirements
- Define and use a dynamic template that satisfies a given set of requirements
- Define an Index Lifecycle Management policy for a time-series index
- Define an index template that creates a new data stream

Searching Data

- Write and execute a search query for terms and/or phrases in one or more fields of an index
- Write and execute a search query that is a Boolean combination of multiple queries and filters
- Write an asynchronous search
- Write and execute metric and bucket aggregations
- Write and execute aggregations that contain sub-aggregations
- Write and execute a query that searches across multiple clusters
- Write and execute a search that utilizes a runtime field

Developing Search Applications

- Highlight the search terms in the response of a query
- Sort the results of a query by a given set of requirements
- Implement pagination of the results of a search query
- Define and use index aliases
- Define and use a search template

Data Processing

- Define a mapping that satisfies a given set of requirements
- Define and use a custom analyzer that satisfies a given set of requirements
- Define and use multi-fields with different data types and/or analyzers

- Use the Reindex API and Update By Query API to reindex and/or update documents
- Define and use an ingest pipeline that satisfies a given set of requirements, including the use of Painless to modify documents
- Define runtime fields to retrieve custom values using Painless scripting

Cluster Management

- Diagnose shard issues and repair a cluster's health
- Backup and restore a cluster and/or specific indices
- Configure a snapshot to be searchable
- Configure a cluster for cross-cluster search
- Implement cross-cluster replication

Assumptions

This is a very barebones guide. It will list the various topics and tasks that need to be done with a brief explanation of the example, a potential solution, an explanation for certain concepts, optional clean-up since you will be creating a number of resources in your Elastic cluster, and a list of documentation specific to the solution that should help you to understand why the solution was recommended.

The study guide examples assume you have a foundational understanding of search, search technologies, and Elasticsearch. It is:

- Not an introductory text on search and search technologies.
- Not an Elastic or Kibana tutorial.
- Not a JSON tutorial.

The examples are presented as REST API calls in JSON for the **Elastic Kibana Console**. In the **Things You Must Do** section, we will show you how to translate these REST API calls into `curl` commands. This is to ensure you understand how to execute the calls both ways, but `curl` commands will not be used in the examples.

Things You Must Do

Regardless of where you are running Elastic (locally or from the Elastic Cloud), you will need two pieces of information to access your deployment:

- Username
- Password

How and where you obtain these will depend on whether you have a local instance of Elasticsearch/Kibana or are using an Elastic Cloud deployment.

Instructions for installing a local instance of Elasticsearch/Kibana can be found in the **Appendix**, along with basic instructions on obtaining the username/password when you create an Elastic Cloud deployment. These instructions are also available in the Elastic documentation.

If you decide to run these examples from the command line using `curl`, you must:

- Have `curl` installed.
- Have your Elasticsearch username (`elastic`) and password handy. The password will vary depending on whether you are running your own local copy of the Elastic Stack or using the Elastic Cloud.
- If you are running `curl` against your **local** container instance, your command line should look like this:

```
curl --cacert http_ca.crt -u elastic:[container instance password here] -X
↪ [desired HTTP command] "https://[elastic endpoint here]/[API path as
↪ appropriate]"
```

The `http_ca.crt` file should be extracted from the container during deployment. If you haven't done this, execute the following command in the location where you are doing your certification work (assuming your Elasticsearch container is named **es01**):

```
docker cp es01:/usr/share/elasticsearch/config/certs/http_ca.crt .
```

Keep that file secret. Keep that file safe.

As a test, you should be able to run:

```
curl --cacert http_ca.crt -u elastic:[container instance password here]
↪ https://localhost:9200/
```

You should get some reasonable output.

- If you are running `curl` against the **Elastic Cloud**, your command line should look like this:

```
curl -u elastic:[elastic cloud deployment password here] -X [desired HTTP
↪ command] "https://[elastic endpoint here]/[API path as appropriate]"
```

The only difference between the above and the local command is that you don't need the certificate file if you are running `curl` against the Elastic Cloud.

Originally, the examples included both the REST API calls and the `curl` commands. Since the `curl` commands differ slightly between the local instance and the Elastic Cloud instance, they have been left out. If you want to run `curl` using the Elastic REST API, remember that the REST API looks like this in the Elastic Cloud console:

```
GET /
```

or

```
PUT /example_index
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text"
      },
      "description": {
        "type": "text"
      }
    }
  }
}
```

The associated `curl` command would look like this:

- Local

The first call:

```
curl --cacert http_ca.crt -u elastic:[container instance password here] -X
 ↪  GET https://localhost:9200/
```

or for the second call (note the use of single quotes around the JSON):

```
curl --cacert http_ca.crt -u elastic:[container instance password here] -X
 ↪  PUT https://localhost:9200/example_index -H 'Content-Type:
 ↪  application/json' -d'
{
```

```
  "mappings": {
    "properties": {
      "title": {
        "type": "text"
      },
      "description": {
        "type": "text"
      }
    }
  }
}'
```

- Elastic Cloud

The first call:

```
curl -u elastic:[elastic cloud deployment password here] -X GET
 ↪  "https://[elastic endpoint here]/"
```

or for the second call (note the use of single quotes around the JSON):

```
curl -u elastic:[elastic cloud deployment password here] -X PUT "[elastic
 ↪  endpoint here]" -H 'Content-Type: application/json' -d'
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text"
      },
      "description": {
        "type": "text"
      }
    }
  }
}'
```

How to Pass the Test

Do the examples and more. Be over-prepared.

Go through the documentation so you know where to look when a task shows up in the certification exam and you are not sure what the syntax or format of a given JSON might be. This is an open book test, but the only book you can use is the Elastic documentation.

Learn the basics. The Elastic console has code completion, so you don't have to remember everything—just what might be the appropriate JSON elements for the solution you are implementing.

Learn the query syntax of Query DSL. This is a search engine, after all, and knowledge of querying is fundamental to the certification exam.

While there is no magic bullet, the exam should not be that hard if you already have knowledge of:

- Search and search technologies (preferably hands-on)
- JSON

Things that will trip you up:

- Syntax
- Depth of various JSON nodes

Summary

This book will not teach you Elasticsearch or Kibana or any of the other Elastic products available in the first half of 2024. It assumes you have an understanding of various search topics, JSON, REST APIs, and areas like regular expressions and web technologies. The examples are valid with the current documentation and were all confirmed as working in the same time period.

If you run into any problems with the examples, please send an email to support@brushedsteelconsulting.com. When in doubt, asking in the Elastic community or one of the many public LLMs available should help as well:

- ChatGPT
- Claude.ai
- Gemini
- Meta.ai
- Perplexity.ai

All of the examples were originally generated by the various LLMs listed above with many changes made by an actual human as the examples and the generated content left much to be desired.

Disclaimer: this book was written with the assistance of various tools, including a host of LLMs, but always under the guidance of the author. Make of that what you will.

1 Data Management

1.1 Task: Define an index that satisfies a given set of requirements

Example 1: Creating an Index for a Blogging Platform

Requirements

- The platform hosts articles, each with `text` content, a publication date, author details, and tags.
- Articles need to be searchable by `content`, `title`, and `tags`.
- The application requires fast search responses and efficient storage.
- The application should handle date-based queries efficiently.
- Author details are nested objects that include the author's name and email.

Steps

(1) **Open the Kibana Console** or use a REST client.

(2) **Define Mappings:**

- **Content and Title:** Use the `text` data type
- **Publication Date:** Use the `date` data type
- **Tags:** Use the `keyword` data type for exact matching
- **Author:** Use a `nested` object to keep author details searchable and well-structured

(3) Create the index

```
PUT blog_articles
{
  "settings": {
    "number_of_shards": 3,
    "number_of_replicas": 1
  },
  "mappings": {
    "properties": {
      "title": {
```

```
        "type": "text"
      },
      "content": {
        "type": "text"
      },
      "publication_date": {
        "type": "date"
      },
      "author": {
        "type": "nested",
        "properties": {
          "name": {
            "type": "text"
          },
          "email": {
            "type": "keyword"
          }
        }
      },
      "tags": {
        "type": "text"
      }
    }
  }
}
```

Or insert the settings and mappings separately.

```
PUT blog_articles
{
  "settings": {
    "number_of_shards": 3,
    "number_of_replicas": 1
  }
}
```

And:

```
PUT blog_articles/_mapping
{
  "properties": {
    "title": {
      "type": "text"
```

```
    },
    "content": {
      "type": "text"
    },
    "publication_date": {
      "type": "date"
    },
    "author": {
      "type": "nested",
      "properties": {
        "name": {
          "type": "text"
        },
        "email": {
          "type": "keyword"
        }
      }
    },
    "tags": {
      "type": "text"
    }
  }
}
```

Test

1. Verify the index

```
GET /_cat/indices
```

2. Verify the mappings

```
GET /blog_articles/_mapping
```

3. Index and search for a document

```
# Index
POST /blog_articles/_doc
{
  "title" : "My First Blog Post",
  "content" : "What an interesting way to go...",
  "publication_date" : "2024-05-15",
```

```
  "tags" : "superb",
  "author" : {
    "name" : "John Doe",
    "email" : "john@doe.com"
  }
}
```

```
# Search like this
GET /blog_articles/_search
```

```
# Or search like this
GET /blog_articles/_search?q=tags:superb
```

```
# Or search like this
GET blog_articles/_search
{
  "query": {
    "query_string": {
      "default_field": "tags",
      "query": "superb"
    }
  }
}
```

```
# Or search like this
GET blog_articles/_search
{
  "query": {
    "nested": {
      "path": "author",
      "query": {
        "match": {
          "author.name": "john"
        }
      }
    }
  }
}
```

Considerations

- **Shards and Replicas:** Adjust these settings based on expected data volume and query load.
- **Nested Objects:** These are crucial for maintaining the structure and searchability of complex data like author details.

Clean-up (optional)

- In the console execute the following

```
DELETE blog_articles
```

Documentation

- cat indices API
- Create Index API
- Get mapping API
- Search API

Example 2: Creating an Index for Log Data

Requirements

1. Store log data with a timestamp field

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the index

```
PUT /log_data
{
  "settings": {
    "number_of_shards": 3
  },
  "mappings": {
    "properties": {
      "@timestamp": {
        "type": "date"
      },
      "log_source": {
        "type": "keyword"
      },
      "message": {
        "type": "text"
      }
    }
  }
}
```

Test

1. Verify the index creation

```
GET /log_data
```

Or

```
GET /_cat/indices
```

2. Verify the field mapping

```
GET /log_data/_mapping
```

3. Index and search for a sample document

    1. Index

```
PUT /log_data/_doc/1
{
  "@timestamp": "2023-05-16T12:34:56Z",
  "log_source": "web_server",
  "message": "HTTP request received"
}
```

    2. Search

```
GET /log_data/_search
```

    The response should show the indexed document.

Considerations

- In `settings`, `number_of_replicas` doesn't appear as its default is set to one 1 which is sufficient. The field `number_of_shards` should be higher than 1 depending on the requirements for a log index. No, you do not need to have a settings block for the index to be created.
- The `@timestamp` field is mapped as a date type for time-based data management.
- The `log_source` field is mapped as a `keyword` type to enable custom routing based on its value.

Clean-up (optional)

- In the console execute the following

```
DELETE log_data
```

Documentation

- cat indices API
- Create Index API
- Get mapping API

- Search API

Example 3: Creating an index for e-commerce product data with daily updates

Requirements

1. Store product information including `name`, `description`, `category`, `price`, and `stock_level`.
2. Allow filtering and searching based on product `name`, `category`, and `price` range.
3. Enable aggregations to calculate average price per category.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Define mappings:
    - Use the `text` data type for `name` and `description` to allow full-text search.
    - Use the `keyword` data type for `category` to enable filtering by exact terms.
    - Use the `integer` data type for `price` to allow for range queries and aggregations.
    - Use the `integer` data type for `stock_level` for inventory management.
3. Create the index

```
PUT products
{
  "mappings": {
    "properties": {
      "name": { "type": "text" },
      "description": { "type": "text" },
      "category": { "type": "keyword" },
      "price": { "type": "integer" },
      "stock_level": { "type": "integer" }
    }
  }
}
```

- **Configure analyzers (optional):**
    - You can define custom analyzers for `name` and `description` to handle special characters or stemming based on your needs. Notice two things:
        1. How the `custom_analyzer` refers to the `filter` and `tokenizer` (both of which are optional).

2. The fields that will use `custom_analyzer`, `name` and `description`, have an analyzer reference to `custom_analyzer`.

```
PUT /products
{
  "settings": {
    "analysis": {
      "tokenizer": {
        "custom_tokenizer": {
          "type": "standard"
        }
      },
      "filter": {
        "custom_stemmer": {
          "type": "stemmer",
          "name": "english"
        },
        "custom_stop": {
          "type": "stop",
          "stopwords": "_english_"
        }
      },
      "analyzer": {
        "custom_analyzer": {
          "type": "custom",
          "tokenizer": "custom_tokenizer",
          "filter": [
            "lowercase",
            "custom_stop",
            "custom_stemmer"
          ]
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "name": {
        "type": "text",
        "analyzer": "custom_analyzer"
      },
      "description": {
        "type": "text",
```

```
        "analyzer": "custom_analyzer"
      },
      "category": {
        "type": "keyword"
      },
      "price": {
        "type": "integer"
      },
      "stock_level": {
        "type": "integer"
      }
    }
  }
}
```

Test

1. Verify the index creation

```
GET products
```

   Or

```
GET /_cat/indices
```

2. Verify the field mapping

```
GET /products/_mapping
```

3. Index and search some sample product data

   1. Index some products

```
POST /products/_bulk
{ "index": { "_index": "products", "_id": "1" } }
{ "name": "Wireless Bluetooth Headphones", "description":
 ↪  "High-quality wireless Bluetooth headphones with
 ↪  noise-cancellation and long battery life.", "category":
 ↪  "electronics", "price": 99, "stock_level": 250 }
{ "index": { "_index": "products", "_id": "2" } }
{ "name": "Stainless Steel Water Bottle", "description": "Durable
 ↪  stainless steel water bottle, keeps drinks cold for 24 hours and
 ↪  hot for 12 hours.", "category": "home", "price": 25,
 ↪  "stock_level": 500 }
```

```
{ "index": { "_index": "products", "_id": "3" } }
{ "name": "Smartphone", "description": "Latest model smartphone with high-resolution display and fast processor.", "category": "electronics", "price": 699, "stock_level": 150 }
{ "index": { "_index": "products", "_id": "4" } }
{ "name": "LED Desk Lamp", "description": "Energy-efficient LED desk lamp with adjustable brightness and flexible neck.", "category": "home", "price": 45, "stock_level": 300 }
{ "index": { "_index": "products", "_id": "5" } }
{ "name": "4K Ultra HD TV", "description": "55-inch 4K Ultra HD TV with HDR support and smart features.", "category": "electronics", "price": 499, "stock_level": 200 }
{ "index": { "_index": "products", "_id": "6" } }
{ "name": "Vacuum Cleaner", "description": "High-suction vacuum cleaner with multiple attachments for versatile cleaning.", "category": "home", "price": 120, "stock_level": 100 }
```

2. Search

```
GET /products/_search?q=name:desk
```

4. Use aggregations to calculate the average price per category.

```
POST /products/_search
{
  "size": 0,
  "aggs": {
    "average_price_per_category": {
      "terms": {
        "field": "category"
      },
      "aggs": {
        "average_price": {
          "avg": {
            "field": "price"
          }
        }
      }
    }
  }
}
```

Considerations

- Using the appropriate data types ensures efficient storage and querying capabilities.
- Text fields allow full-text search, while keyword fields enable filtering by exact terms.

Clean-up (optional)

- In the console execute the following

```
DELETE products
```

Documentation

- Aggregations
- cat indices API
- Create Index API
- Get mapping API
- Search API

1.2 Task: Define and use an index template for a given pattern that satisfies a given set of requirements

Example 1: Creating an index template for a user profile data

Requirements

- Create an index template named `user_profile_template`.
- The template should apply to indices starting with `user_profile-`.
- The template should have two shards and one replica.
- The template should have a mapping for the `name` field as a `text` data type with an analyzer of `standard`.
- The template should have a mapping for the `age` field as an `integer` data type.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the index template

```
PUT /_index_template/user_profile_template
{
  "index_patterns": ["user_profile-*"],
  "template": {
    "settings": {
      "number_of_shards": 2,
      "number_of_replicas": 1
    },
    "mappings": {
      "properties": {
        "name": {
          "type": "text",
          "analyzer": "standard"
        },
        "age": {
          "type": "integer"
        }
      }
    }
  }
}
```

Test

1. Verify the index template was created

```
GET _index_template/user_profile_template
```

2. Create an index named `user_profile-2024` using the REST API:

```
PUT /user_profile_2024
```

3. Verify that the index was created with the expected settings and mappings:

```
GET /user_profile_2024/_settings
```

```
GET /user_profile_2024/_mapping
```

Considerations

- Two shards are chosen to allow for parallel processing and improved search performance.
- One replica is chosen for simplicity and development purposes; in a production environment, this would depend on the expected data volume and search traffic.
- The `standard` analyzer is chosen for the `name` field to enable standard text analysis.

Clean-up (optional)

- In the console execute the following

```
DELETE /user_profile_2024
DELETE /_index_template/user_profile_template
```

Documentation

- Analyzers
- Create Index API
- Index templates

Example 2: Creating a monthly product index template

Requirements

- Index name pattern: `products-*`
- Index settings:
  - Number of shards: 3
  - Number of replicas: 2
- Mapping:
  - Field `name` should be of type `text`
  - Field `description` should be of type `text`
  - Field `price` should be of type `float`
  - Field `category` should be of type `keyword`

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the index template

```
PUT _template/monthly_products
{
  "index_patterns": ["products-*"],
  "settings": {
    "number_of_shards": 3,
    "number_of_replicas": 2
  },
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "description": {
        "type": "text"
      },
      "price": {
        "type": "float"
      },
      "category": {
        "type": "keyword"
      }
```

```
    }
  }
}
```

Test

1. Verify the index template was created

```
GET _index_template/monthly_products
```

2. Create a new index matching the pattern (e.g., **products-202305**):

```
PUT products-202305
```

3. Verify that the index was created with the expected settings and mappings:

```
GET /products-202305/_settings
```

```
GET /products-202305/_mapping
```

4. Index a sample document and verify that the mapping is applied correctly:

    1. Index

    ```
    POST products-202305/_doc
    {
      "name": "Product A",
      "description": "This is a sample product",
      "price": 19.99,
      "category": "Electronics"
    }
    ```

    2. Search

    ```
    GET products-202305/_search
    ```

The response should show the correct mapping for the fields specified in the index template.

Considerations

- The `index_patterns` field specifies the pattern for index names to which this template should be applied.

- The `number_of_shards` and `number_of_replicas` settings are chosen based on the expected data volume and high availability requirements.
- The `text` type is used for `name` and `description` fields to enable full-text search and analysis.
- The `float` type is used for the `price` field to support decimal values.
- The `keyword` type is used for the `category` field to prevent analysis and treat the values as exact matches.

Clean-up (optional)

- In the console execute the following

```
DELETE products-202305
```

```
DELETE _template/monthly_products
```

Documentation

- Create Index API
- Index templates
- Search API

Example 3: Creating an index template for log indices

Requirements

- The template should apply to any index starting with `logs-`.
- The template must define settings for three primary shards and one replica.
- The template should include mappings for fields `@timestamp`, `log_level`, and `message`.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the index template

```
PUT /_index_template/logs_template
{
  "index_patterns": ["logs-*"],
  "template": {
    "settings": {
      "index": {
        "number_of_shards": 3,
        "number_of_replicas": 1
      }
    },
    "mappings": {
      "properties": {
        "@timestamp": {
          "type": "date"
        },
        "log_level": {
          "type": "keyword"
        },
        "message": {
          "type": "text"
        }
      }
    }
  }
}
```

Test

1. Verify the index template was created

```
GET _index_template/logs_template
```

2. Create a new index matching the pattern (e.g., **logs-202405**)

```
PUT logs-202405
```

3. Verify that the index was created with the expected settings and mappings

```
GET /logs-202405/_settings
```

```
GET /logs-202405/_mapping
```

4. Index a sample document and verify that the mapping is applied correctly:

    1. Index

```
POST logs-202405/_doc
{
  "@timestamp": "2024-05-16T12:34:56Z",
  "log_level": "ERROR",
  "message": "Help!"
}
```

    2. Search

```
GET logs-202405/_search
```

The response should show the correct mapping for the fields specified in the index template.

Considerations

- **Index Patterns:** The template applies to any index starting with `logs-`, ensuring consistency across similar indices.
- **Number of Shards:** Three shards provide a balance between performance and resource utilization.
- **Replicas:** A single replica ensures high availability and fault tolerance.
- **Mappings:** Predefined mappings ensure that the fields are properly indexed and can be efficiently queried.

Clean-up (optional)

- In the console execute the following

```
DELETE logs-202405
```

```
DELETE _index_template/logs_template
```

Documentation

- Create Index API
- Logstash: Event Dependent Configuration
- Index templates
- Search API

1.3 Task: Define and use a dynamic template that satisfies a given set of requirements

FYI: The difference between **index templates** and **dynamic templates** is:

*An **index template** is a way to define settings, mappings, and other configurations that should be applied automatically to new indices when they are created.* A **dynamic template** is part of the mapping definition within an index template or index mapping that allows Elasticsearch to dynamically infer the mapping of fields based on field names, data patterns, or the data type detected.

There is one example per field mapping type. They all use an explicit dynamic template, but Exercise 1 also shows the use of a dynamic template embedded in the index definition.

Example 1: Create a Dynamic Template for Logging Using Field Name Patterns

Requirements

- Apply a specific text analysis to all fields that end with `_log`.
- Use a `keyword` type for all fields that start with `status_`.
- Default to `text` with a standard analyzer for other string fields.
- Define a custom `log_analyzer` for `_log` fields.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Define the dynamic template

- As part of the index definition

```
PUT /logs_index
{
  "mappings": {
    "dynamic_templates": [
      {
        "log_fields": {
          "match": "*_log",
          "mapping": {
            "type": "text",
            "analyzer": "log_analyzer"
          }
        }
```

```
      },
      {
        "status_fields": {
          "match": "status_*",
          "mapping": {
            "type": "keyword"
          }
        }
      },
      {
        "default_string": {
          "match_mapping_type": "string",
          "mapping": {
            "type": "text",
            "analyzer": "standard"
          }
        }
      }
    ]
  },
  "settings": {
    "analysis": {
      "analyzer": {
        "log_analyzer": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": ["lowercase", "stop"]
        }
      }
    }
  }
}
```

– or as a standalone definition to be added to indexes as needed using the index_pattern

```
PUT /_index_template/logs_dyn_template
{
  "index_patterns": ["logs_*"],
  "template": {
    "mappings": {
      "dynamic_templates": [
```

```
        {
          "log_fields": {
            "match": "*_log",
            "mapping": {
              "type": "text",
              "analyzer": "log_analyzer"
            }
          }
        },
        {
          "status_fields": {
            "match": "status_*",
            "mapping": {
              "type": "keyword"
            }
          }
        },
        {
          "default_string": {
            "match_mapping_type": "string",
            "mapping": {
              "type": "text",
              "analyzer": "standard"
            }
          }
        }
      ]
    },
    "settings": {
      "analysis": {
        "analyzer": {
          "log_analyzer": {
            "type": "custom",
            "tokenizer": "standard",
            "filter": ["lowercase", "stop"]
          }
        }
      }
    }
  }
}
```

Test

1. Verify the dynamic template was created
    - If you used the embedded version

```
GET /logs_index/_mapping
```

    - If you used the standalone version

```
GET /_index_template/logs_dyn_template
```

2. Create a new index matching the pattern (e.g., logs-202405)
    - Optional if you used the embedded version

```
PUT logs_index
```

3. Verify that the created index has the expected settings and mappings
    - Ensure error_log is of type text with log_analyzer
    - Ensure status_code is of type keyword
    - Ensure message is of type text with standard analyzer

```
GET /logs_index/_mapping
```

4. Index a sample document and verify that the mapping is applied correctly

```
POST /logs_index/_doc/1
{
  "error_log": "This is an error log message.",
  "status_code": "200",
  "message": "Regular log message."
}
```

5. Perform Searches:
    - Search within error_log and verify the custom analyzer is applied

```
GET /logs_index/_search
{
  "query": {
    "match": {
      "error_log": "error"
    }
```

```
  }
}
```

- Check if status_code is searchable as a keyword

```
GET /logs_index/_search
{
  "query": {
    "term": {
      "status_code": "200"
    }
  }
}
```

Considerations

- The custom analyzer log_analyzer is used to provide specific tokenization and filtering for log fields.
- The keyword type for status_* fields ensures they are treated as exact values, useful for status codes.
- The default_string template ensures other string fields are analyzed with the standard analyzer, providing a balanced default.

Clean-up (optional)

- Delete the index

```
DELETE logs_index
```

- Delete the dynamic template

```
DELETE /_index_template/logs_dyn_template
```

Documentation

- Dynamic Templates
- Custom Analyzers
- Put Mapping API
- Index API
- Search API

Example 2: Create Dynamic Template for Data Types

Requirements

- All string fields should be treated as `text` with a `standard` analyzer.
- All long fields should be treated as `integer`.
- All `date` fields should use a specific date format.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Define the Dynamic Template

```
PUT /_index_template/data_type_template
{
  "index_patterns": ["data_type_*"],
  "template": {
    "mappings": {
      "dynamic_templates": [
        {
          "strings_as_text": {
            "match_mapping_type": "string",
            "mapping": {
              "type": "text",
              "analyzer": "standard"
            }
          }
        },
        {
          "longs_as_integer": {
            "match_mapping_type": "long",
            "mapping": {
              "type": "integer"
            }
          }
        },
        {
          "dates_with_format": {
            "match_mapping_type": "date",
            "mapping": {
              "type": "date",
              "format": "yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"
```

```
            }
          }
        }
      ]
    }
  }
}
```

Test

1. Verify the dynamic template was created

```
GET /_index_template/data_type_template
```

2. Create a new index matching the pattern

```
PUT data_type_202405
```

3. Check the Field Types

   - Verify that all string fields are mapped as `text` with the `standard` analyzer.
   - Verify that all long fields are mapped as `integer`.
   - Verify that all `date` fields are mapped with the correct format.

```
GET /data_type_202405/_mapping
```

4. Insert sample documents to ensure that the dynamic template is applied correctly

```
POST /data_type_202405/_bulk
{ "index": { "_index": "data_type_202405", "_id": "1" } }
{ "name": "Wireless Bluetooth Headphones", "release_date":
 ↪  "2024-05-28T14:35:00.000Z", "price": 99 }
{ "index": { "_index": "data_type_202405", "_id": "2" } }
{ "description": "Durable stainless steel water bottle", "launch_date":
 ↪  "2024-05-28T15:00:00.000Z", "quantity": 500 }
```

5. Perform Searches

   - Search launch_date

```
GET /data_type_202405/_search
{
  "query": {
```

```
    "query_string": {
      "query": "launch_date:\"2024-05-28T15:00:00.000Z\""
    }
  }
}
```

- Check if price is searchable as a value

```
GET /data_type_202405/_search
{
  "query": {
    "query_string": {
      "query": "price: 99"
    }
  }
}
```

Considerations

- **Dynamic Templates:** Using dynamic templates based on data types allows for flexible and consistent field mappings without needing to know the exact field names in advance.
- **Data Types:** Matching on data types (string, long, date) ensures that fields are mapped appropriately based on their content.
- **Date Format:** Specifying the date format ensures that date fields are parsed correctly, avoiding potential issues with date-time representation.

Clean-up (optional)

- Delete the index

```
DELETE data_type_202405
```

- Delete the dynamic template

```
DELETE /_index_template/data_type_template
```

Documentation

- Dynamic Templates
- Mapping

- Analyzers

Example 3: Create a Dynamic Template for Logging Data for Data Patterns

Requirements

- Automatically map fields that end with "_ip" as IP type.
- Map fields that start with "timestamp_" as date type.
- Map any field containing the word "keyword" as a keyword type.
- Use a custom analyzer for fields ending with "_text".

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the dynamic template

```
PUT /_index_template/logs_template
{
  "index_patterns": ["logs*"],
  "template": {
    "settings": {
      "analysis": {
        "analyzer": {
          "custom_analyzer": {
            "type": "standard",
            "stopwords": "_english_"
          }
        }
      }
    },
    "mappings": {
      "dynamic_templates": [
        {
          "ip_fields": {
            "match": "*_ip",
            "mapping": {
              "type": "ip"
            }
          }
        },
        {
          "date_fields": {
            "match": "timestamp_*",
            "mapping": {
```

```
            "type": "date"
          }
        }
      },
      {
        "keyword_fields": {
          "match": "*keyword*",
          "mapping": {
            "type": "keyword"
          }
        }
      },
      {
        "text_fields": {
          "match": "*_text",
          "mapping": {
            "type": "text",
            "analyzer": "custom_analyzer"
          }
        }
      }
    ]
  }
}
}
```

Test

1. Verify the dynamic template was created

```
GET /_index_template/logs_template
```

2. Create a new index matching the pattern

```
PUT logs_202405
```

3. Check the Field Types

   - Verify that all `_ip` fields are mapped as `ip`
   - Verify that all `timestamp_` fields are mapped as `date`
   - Verify that all fields that contain the string keyword are mapped as `keyword`

```
GET /logs_202405/_mapping
```

4. Insert sample documents to ensure that the dynamic template is applied correctly

```
POST /logs_202405/_bulk
{ "index": { "_id": "1" } }
{ "source_ip": "192.168.1.1", "timestamp_event": "2024-05-28T12:00:00Z",
 ↪  "user_keyword": "elastic", "description_text": "This is a log
 ↪  entry." }
{ "index": { "_id": "2" } }
{ "destination_ip": "10.0.0.1", "timestamp_access":
 ↪  "2024-05-28T12:05:00Z", "log_keyword": "search", "details_text":
 ↪  "Another log entry." }
```

5. Perform Searches

   - Search source_ip

```
GET /logs_202405/_search
{
  "query": {
    "query_string": {
      "query": "source_ip:\"192.168.1.1\""
    }
  }
}
```

   - Check if timestamp_event is searchable as a date

```
GET /logs_202405/_search
{
  "query": {
    "query_string": {
      "query": "timestamp_event:\"2024-05-28T12:00:00Z\""
    }
  }
}
```

Considerations

- The use of patterns in the dynamic template ensures that newly added fields matching the criteria are automatically mapped without the need for manual intervention.

- Custom analyzer configuration is critical for ensuring text fields are processed correctly, enhancing search capabilities.

Clean-up (optional)

- Delete the index

```
DELETE logs_202405
```

- Delete the dynamic template

```
DELETE /_index_template/logs_template
```

Documentation

- Dynamic Templates
- Mapping Types
- Analyzers

1.4 Task: Define an Index Lifecycle Management policy for a timeseries index

Example 1: Creating an ILM policy for log data indices

Requirements

- Indices are prefixed with `logstash-`
- Indices should be rolled over daily (create a new index every day).
- Old indices should be deleted after 30 days.

Steps using the Elastic/Kibana UI

1. Open the hamburger menu and click on **Management > Data > Life Cycle Policies**.
2. Press **+ Create New Policy**.
3. Enter the following:
   - Policy name: logstash-example-policy.
   - **Hot** phase:
     - Change **Keep Data in the Phase Forever** (the infinity icon) to **Delete Data After This Phase** (the trashcan icon).
     - Click **Advanced Settings**.
     - Unselect **Use Recommended Defaults**.
     - Set **Maximum Age** to **1**.
   - **Delete** phase:
     - Move data into phase when: 30 days old.
4. Press **Save Policy**.
5. **Open the Kibana Console** or use a REST client.
6. Create an index template that will match on indices that match the pattern `logstash-*`.

```
PUT /_index_template/ilm_logstash_index_template
{
  "index_patterns": ["logstash-*"]
}
```

7. Return to the **Management > Data > Life Cycle Policies** page.
8. Press the plus sign (+) to the right of logstash-example-policy.
   1. The **Add Policy "logstash-example-policy" to index template** dialog opens.

2. Click on the **Index Template** input field and type the first few letters of the index template created above.
3. Select the template created above (ilm_logstash_index_template).
4. Press **Add Policy**.

9. **Open the Kibana Console** or use a REST client.

10. List `ilm_logs_index_template`. Notice the ILM policy is now part of the index template.

```
GET /_index_template/ilm_logstash_index_template
```

Output from the `GET`:

```
{
  "index_templates": [
    {
      "name": "ilm_logstash_index_template",
      "index_template": {
        "index_patterns": ["logstash-*"],
        "template": {
          "settings": {
            "index": {
              "lifecycle": {
                "name": "logstash-example-policy"
              }
            }
          }
        },
        "composed_of": []
      }
    }
  ]
}
```

11. Create an index.

```
PUT logstash-2024.05.16
```

12. Verify the policy is there.

```
GET logstash-2024.05.16
```

The output should look something like this:

```
{
  "logstash-2024.05.16": {
    "aliases": {},
    "mappings": {},
    "settings": {
      "index": {
        "lifecycle": {
          "name": "logstash-example-policy"
        },
        "routing": {
          "allocation": {
            "include": {
              "_tier_preference": "data_content"
            }
          }
        },
        "number_of_shards": "1",
        "provided_name": "logstash-2024.05.16",
        "creation_date": "1717024100387",
        "priority": "100",
        "number_of_replicas": "1",
        "uuid": "mslAKuZGTpSDdFr4hSpAAA",
        "version": {
          "created": "8503000"
        }
      }
    }
  }
}
```

Steps Using the REST API (which I would not recommend)

1. **Open the Kibana Console** or use a REST client.
2. Create the ILM policy.

```
PUT _ilm/policy/logstash-example-policy
{
  "policy": {
    "phases": {
      "hot": {
        "actions": {
```

```
          "rollover": {
            "max_age": "1d"
          }
        }
      },
      "delete": {
        "min_age": "30d",
        "actions": {
          "delete": {}
        }
      }
    }
  }
}
```

3. Create an index template that includes the above policy. The two fields within settings are required.

```
PUT /_index_template/ilm_logstash_index_template
{
  "index_patterns": ["logstash-*"],
  "template": {
    "settings": {
      "index.lifecycle.name": "logstash-example-policy",
      "index.lifecycle.rollover_alias": "logstash"
    }
  }
}
```

Test

1. Verify the ILM policy exists in Kibana under **Management > Data > Index Lifecycle Policies**.
2. Verify the Index Lifecycle Management policy exists and references the index template.

```
GET /_ilm/policy/logstash-example-policy
```

3. Verify the policy is referenced in the index template.

```
GET /_index_template/ilm_logstash_index_template
```

4. Create a new index that matches the pattern logstash-*.

```
PUT /logstash-index
```

5. Verify the index has the policy in its definition.

```
GET /logstash-index
```

Considerations

- The index template configures 1 shard and the ILM policy/alias for rollover.
- The rollover action creates a new index when the max_age is reached.
- The delete phase removes indices older than 30 days.

Clean-up (optional)

- Delete the index.

```
DELETE logstash-index
```

- Delete the index template.

```
DELETE /_index_template/ilm_logstash_index_template
```

- Delete the policy.

```
DELETE /_ilm/policy/logstash-example-policy
```

Documentation

- Create Index API
- Create or Update Index Template API
- ILM Settings
- Index Lifecycle Management

Example 2: Creating an ILM policy for logs indices retention for 7, 30 and 90 days

Requirements

- The policy should be named `logs-policy`.
- It should have a hot phase with a duration of 7 days.
- It should have a warm phase with a duration of 30 days.
- It should have a cold phase with a duration of 90 days.
- It should have a delete phase.
- The policy should be assigned to indices matching the pattern `ilm_logs_*`.

Steps using the Elastic/Kibana UI

1. Open the hamburger menu and click on **Management > Data > Life Cycle Policies**.
2. Press **+ Create New Policy**.
3. Enter the following:
   - Policy name: logs-policy.
   - **Hot** phase:
     - Press the garbage can icon to the right to **delete data after this phase**.
   - **Warm** phase:
     - Move data into phase when: 7 days old.
     - Leave **Delete data after this phase**.
   - **Cold** phase:
     - Move data into phase when: 30 days old.
     - Leave **Delete data after this phase**.
   - **Delete** phase:
     - Move data into phase when: 90 days old.
4. Press **Save Policy**.
5. **Open the Kibana Console** or use a REST client.
6. Create an index template that will match on indices that match the pattern `ilm_logs_*`.

```
PUT /_index_template/ilm_logs_index_template
{
  "index_patterns": ["ilm_logs_*"]
}
```

7. Return to the **Management > Data > Life Cycle Policies** page.

8. Press the plus sign (+) to the right of logs_policy.
9. The **Add Policy "logs-policy" to index template** dialog opens.
10. Click on the **Index Template** input field and type the first few letters of the index template created above.
11. Select the template created above (`ilm_logs_index_template`).
12. Press **Add Policy**.
13. **Open the Kibana Console** or use a REST client.
14. List `ilm_logs_index_template`. Notice the ILM policy is now part of the index template.

```
GET /_index_template/ilm_logs_index_template
```

Output from the `GET` (look for the **settings/index/lifecycle** node):

```
{
  "index_templates": [
    {
      "name": "ilm_logs_index_template",
      "index_template": {
        "index_patterns": ["ilm_logs_*"],
        "template": {
          "settings": {
            "index": {
              "lifecycle": {
                "name": "logs-policy"
              }
            }
          }
        },
        "composed_of": []
      }
    }
  ]
}
```

15. List `logs-policy`.

```
GET _ilm/policy/logs-policy
```

In the `in_use_by` node you will see:

```
"in_use_by": {
  "indices": [],
  "data_streams": [],
  "composable_templates": [
    "ilm_logs_index_template"
  ]
}
```

Steps Using the REST API (which I would not recommend)

1. **Open the Kibana Console** or use a REST client.
2. Create the ILM policy.

```
PUT _ilm/policy/logs-policy
{
  "policy": {
    "phases": {
      "hot": {
        "min_age": "0ms",
        "actions": {
          "set_priority": {
            "priority": 100
          }
        }
      },
      "warm": {
        "min_age": "7d",
        "actions": {
          "set_priority": {
            "priority": 50
          }
        }
      },
      "cold": {
        "min_age": "30d",
        "actions": {
          "set_priority": {
            "priority": 0
          }
        }
      },
```

```
      "delete": {
        "min_age": "90d",
        "actions": {
          "delete": {}
        }
      }
    }
  }
}
```

3. Assign the policy to the indices matching the pattern “logs_*“.

```
PUT /_index_template/ilm_logs_index_template
{
  "index_patterns": ["ilm_logs_*"],
  "template": {
    "settings": {
      "index.lifecycle.name": "logs-policy",
      "index.lifecycle.rollover_alias": "logs"
    }
  }
}
```

Test

1. Verify the ILM policy exists in Kibana under **Management > Data > Index Lifecycle Policies**.
2. Verify the Index Lifecycle Management policy exists and references the index template.

```
GET /_ilm/policy/logs-policy
```

3. Verify the policy is referenced in the index template.

```
GET /_index_template/ilm_logs_index_template
```

4. Create a new index that matches the pattern `ilm_logs_*`.

```
PUT /ilm_logs_index
```

5. Verify the index has the policy in its definition.

```
GET /ilm_logs_index
```

Considerations

- The ILM policy will manage the indices matching the pattern `ilm_logs_*`.
- The hot phase will keep the data for **7** days with high priority and rollover.
- The warm phase will keep the data for **30** days with medium priority.
- The cold phase will keep the data for **90** days with low priority.
- The ILM policy will automatically manage the indices based on their age and size.
- The policy can be adjusted based on the needs of the application and the data.

Clean-up (optional)

- Delete the index.

```
DELETE ilm_logs_index
```

- Delete the index template.

```
DELETE _index_template/ilm_logs_index_template
```

- Delete the policy.

```
DELETE _ilm/policy/logs-policy
```

Documentation

- Create Index API
- Create or Update Index Template API
- ILM Settings
- Index Lifecycle Management

Example 3: Creating an ILM policy for sensor data collected every hour, with daily rollover and retention for one month

Requirements

- Create a new index every day for sensor data (e.g., `sensor_data-{date}`).
- Automatically roll over to a new index when the current one reaches a specific size.
- Delete rolled over indices after one month.

Steps using the Elastic/Kibana UI

1. Open the hamburger menu and click on **Management > Data > Life Cycle Policies**.
2. Press **+ Create New Policy**.
3. Enter the following:
    1. Policy name: **sensor-data-policy**
    2. **Hot** phase:
        1. Change **Keep Data in the Phase Forever** (the infinity icon) to **Delete Data After This Phase** (the trashcan icon).
        2. Click **Advanced Settings**.
        3. Unselect **Use Recommended Defaults**.
        4. Set **Maximum Age** to **1**.
        5. Set **Maximum Index Size** to **10**.
    3. **Delete** phase:
        1. Move data into phase when: **30** days old.
4. Press **Save Policy**.
5. **Open the Kibana Console** or use a REST client.
6. Create an index template that will match on indices that match the pattern "sensor_data-*".

```
PUT /_index_template/sensor_data_index_template
{
  "index_patterns": ["sensor_data-*"]
}
```

7. Return to the **Management > Data > Life Cycle Policies** page.
8. Press the plus sign (+) to the right of sensor-data-policy.
9. The **Add Policy "sensor-data-policy" to index template** dialog opens.

1. Click on the **Index Template** input field and type the first few letters of the index template created above.
2. Select the template created above (sensor_data_index_template).
3. Press **Add Policy**.

10. **Open the Kibana Console** or use a REST client.

11. List sensor_data_index_template. Notice the ILM policy is now part of the index template.

```
GET /_index_template/sensor_data_index_template
```

Output from the `GET`:

```
{
  "index_templates": [
    {
      "name": "sensor_data_index_template",
      "index_template": {
        "index_patterns": ["sensor_data-*"],
        "template": {
          "settings": {
            "index": {
              "lifecycle": {
                "name": "sensor-data-policy"
              }
            }
          }
        },
        "composed_of": []
      }
    }
  ]
}
```

12. List sensor-data-policy.

```
GET /_ilm/policy/sensor-data-policy
```

In the `in_use_by` node you will see:

```
"in_use_by": {
  "indices": [],
  "data_streams": [],
  "composable_templates": [
```

```
    "sensor_data_index_template"
  ]
}
```

OR

Steps Using the REST API (which I would not recommend)

1. **Open the Kibana Console** or use a REST client.
2. Define the ILM policy.

```
PUT _ilm/policy/sensor-data-policy
{
  "policy": {
    "phases": {
      "hot": {
        "min_age": "0ms",
        "actions": {
          "rollover": {
            "max_age": "1d",
            "max_size": "10gb"
          }
        }
      },
      "delete": {
        "min_age": "30d",
        "actions": {
          "delete": {}
        }
      }
    }
  }
}
```

3. Assign the policy to the indices matching the pattern “sensor_data-*“.

```
PUT /_index_template/sensor_data_index_template
{
  "index_patterns": ["sensor_data-*"],
  "template": {
    "settings": {
```

```
      "index.lifecycle.name": "sensor-data-policy",
      "index.lifecycle.rollover_alias": "sensor"
    }
  }
}
```

Test

1. Verify the ILM policy exists in Kibana under **Management > Data > Index Lifecycle Policies**.
2. Verify the Index Lifecycle Management policy exists and references the index template.

```
GET /_ilm/policy/sensor-data-policy
```

3. Verify the policy is referenced in the index template.

```
GET /_index_template/sensor_data_index_template
```

4. Create a new index that matches the pattern sensor_data-*.

```
PUT /sensor_data-20240516
```

5. Verify the index has the policy in its definition.

```
GET /sensor_data-20240516
```

Considerations

- The hot phase size threshold determines the frequency of rollovers.
- The delete phase retention period defines how long rolled over data is stored.

Clean-up (optional)

1. Delete the index.

```
DELETE sensor_data-20240516
```

2. Delete the index template.

```
DELETE /_index_template/sensor_data_index_template
```

3. Delete the policy.

```
DELETE /_ilm/policy/sensor-data-policy
```

Documentation

- Create Index API
- Create or Update Index Template API
- ILM Settings
- Index Lifecycle Management

1.5 Task: Define an index template that creates a new data stream

Data streams in Elasticsearch are used for managing time-series data such as logs, metrics, and events. They can handle large volumes of time-series data in an efficient and scalable manner.

An interesting aspect is that the creation of the data stream is pretty trivial. It normally looks like this in the index template that contains it:

```
```json
...
  "data_stream" : {}
...
```
```

Yep, that's it. The defaults take care of most circumstances.

Also, the `data_stream` must be created in an index template as the `data_stream` needs backing indices. Those backing indices are created when an index is created that matches the pattern in `index_patterns` (basically, any index created using the index template acts as an alias to the actual backing indices created).

Example 1: Creating an index template for continuously flowing application logs

Requirements

- Create a new data stream named "app-logs" to store application logs.
- Automatically create new backing indices within the data stream as needed.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Define the index template that will be used by the data stream to create new backing indices.

```
PUT _index_template/app_logs_index_template
{
  "index_patterns": ["app_logs*"],
  "data_stream": {}
}
```

Test

1. Verify the index template creation.

```
GET _index_template/app_logs_index_template
```

2. Confirm there are no indices named app_logs*.

```
GET /_cat/indices
```

3. Mock sending streaming data by just pushing a few documents to the stream. When sending documents using _bulk, they must use **create** instead of **index**. In addition, the documents must have a `@timestamp` field.

```
POST app_logs/_bulk
{ "create":{} }
{ "@timestamp": "2099-05-06T16:21:15.000Z", "message": "192.0.2.42 - -
 ↪  [06/May/2099:16:21:15 +0000] \"GET /images/bg.jpg HTTP/1.0\" 200
 ↪  24736" }
{ "create":{} }
{ "@timestamp": "2099-05-06T16:25:42.000Z", "message": "192.0.2.255 - -
 ↪  [06/May/2099:16:25:42 +0000] \"GET /favicon.ico HTTP/1.0\" 200 3638"
 ↪  }
```

The response will list the name of the automatically created index, which will look something like this:

```
{
  "errors": false,
  "took": 8,
  "items": [
    {
      "create": {
        "_index": ".ds-app_logs-2099.05.06-000001",
        "_id": "OOazyo8BAvAOn4WaAfdD",
        "_version": 1,
        "result": "created",
        "_shards": {
          "total": 2,
          "successful": 1,
          "failed": 0
        },
        "_seq_no": 2,
        "_primary_term": 1,
```

```
        "status": 201
      }
    },
    {
      "create": {
        "_index": ".ds-app_logs-2099.05.06-000001",
        "_id": "Oeazyo8BAvAOn4WaAfdD",
        "_version": 1,
        "result": "created",
        "_shards": {
          "total": 2,
          "successful": 1,
          "failed": 0
        },
        "_seq_no": 3,
        "_primary_term": 1,
        "status": 201
      }
    }
  ]
}
```

Notice the name of the index is `.ds-app_logs-2099.05.06-000001` (it will probably be slightly different for you).

4. Run:

```
GET /_cat/indices
```

You will see the new index listed. This is the backing index created by the data stream.

5. Check for the **app_logs** data stream under **Management > Data > Index Management > Data Streams**.
6. Verify that the documents were indexed.

```
GET app_logs/_search
```

Notice in the results that `_index` has a different name than `app_logs`.

You can also run the following (using the backing index name your cluster created).

```
GET .ds-app_logs-2024.07.25-000001/_search
```

Considerations

- Data streams provide a more efficient way to handle continuously flowing data compared to daily indices. They are created implicitly through the use of index templates, and you must use the _bulk API when streaming data.
- New backing indices are automatically created within the data stream as needed.
- Lifecycle management policies can be applied to data streams for automatic deletion of older backing indices.

Clean-up (optional)

1. Delete the data stream (deleting the data stream will also delete the backing index).

```
DELETE /_data_stream/app_logs
```

2. Delete the index template.

```
DELETE _index_template/app_logs_index_template
```

Documentation

- Index Templates
- Setting Up a Data Stream

Example 2: Creating an index template for continuously flowing application logs with defined fields

Requirements

- The template should apply to any index matching the pattern `logs*`.
- The template must create a data stream.
- The template should define settings for two primary shards and one replica.
- The template should include mappings for fields `@timestamp`, `log_level`, and `message`.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the index template

```
PUT _index_template/log_application_index_template
{
  "index_patterns": ["logs*"],
  "data_stream": {},
  "template": {
    "settings": {
      "number_of_shards": 2,
      "number_of_replicas": 1
    },
    "mappings": {
      "properties": {
        "@timestamp": {
          "type": "date"
        },
        "log_level": {
          "type": "keyword"
        },
        "message": {
          "type": "text"
        }
      }
    }
  }
}
```

Test

- Verify the index template creation

```
GET _index_template/log_application_index_template
```

- Confirm there are no indices named logs*

```
GET /_cat/indices
```

- Index documents into the data stream

```
POST /logs/_doc
{
  "@timestamp": "2024-05-16T12:34:56",
  "log_level": "info",
  "message": "Test log message"
}
```

This will return a result with the name of the backing index

```
{
  "_index": ".ds-logs-2024.05.16-000001", // yours will be different
  "_id": "PObWyo8BAvAOn4WaC_de",
  "_version": 1,
  "result": "created",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "_seq_no": 0,
  "_primary_term": 1
}
```

Run

```
GET /_cat/indices
```

The index will be listed.

- Confirm the configuration of the backing index matches the index template (your backing index name will be different)

```
GET .ds-logs-2024.05.16-000001
```

- Run a search for the document that was indexed

```
GET .ds-logs-2024.05.16-000001/_search
```

Considerations

- Data streams provide a more efficient way to handle continuously flowing data compared to daily indices. They are created implicitly through the use of index templates, and you must use the _bulk API when streaming data.
- New backing indices are automatically **created** within the data stream as needed.
- Lifecycle management policies can be applied to data streams for automatic deletion of older backing indices (not shown but there is an example at Set Up a Data Stream).

Clean-up (optional)

- Delete the data stream (deleting the data stream will also delete the backing index)

```
DELETE _data_stream/logs
```

- Delete the index template

```
DELETE _index_template/log_application_index_template
```

Documentation

- Index Templates
- Setting Up a Data Stream

Example 3: Creating a metrics data stream for application performance monitoring

Requirements

- Create an index template named `metrics_template`.
- The template should create a new data stream for indices named `metrics-{suffix}`.
- The template should have one shard and one replica.
- The template should have a mapping for the `metric` field as a `keyword` data type.
- The template should have a mapping for the `value` field as a `float` data type.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the index template.

```
PUT _index_template/metrics_template
{
  "index_patterns": ["metrics-*"],
  "data_stream": {},
  "template": {
    "settings": {
      "number_of_shards": 1,
      "number_of_replicas": 1
    },
    "mappings": {
      "properties": {
        "metric": {
          "type": "keyword"
        },
        "value": {
          "type": "float"
        }
      }
    }
  }
}
```

Test

1. Verify the index template creation.

```
GET _index_template/metrics_template
```

2. Confirm there are no indices named **metrics-***.

```
GET /_cat/indices
```

3. Index documents into the data stream.

```
POST /metrics-ds/_doc
{
  "@timestamp": "2024-05-16T12:34:56",
  "metric": "cpu",
  "value": 0.5
}
```

Notice the use of the `@timestamp` field. That is required for any documents going into a data stream.

4. This will return a result with the name of the backing index.

```
{
  "_index": ".ds-metrics-ds-2024.05.16-000001", // yours will be
   ↪  different
  "_id": "P-YFy48BAvAOn4WaUvef",
  "_version": 1,
  "result": "created",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "_seq_no": 1,
  "_primary_term": 1
}
```

5. Run:

```
GET /_cat/indices
```

6. The index will be listed.

7. Confirm the configuration of the backing index matches the index template (your backing index name will be different).

```
GET .ds-metrics-ds-2024.05.16-000001
```

8. Run a search for the document that was indexed.

```
GET .ds-metrics-ds-2024.05.16-000001/_search
```

Considerations

- The `keyword` data type is chosen for the `metric` field to enable exact matching and filtering.
- The `float` data type is chosen for the `value` field to enable precise numerical calculations.
- One shard and one replica are chosen for simplicity and development purposes; in a production environment, this would depend on the expected data volume and search traffic.

Clean-up (optional)

- Delete the data stream (deleting the data stream will also delete the backing index).

```
DELETE _data_stream/metrics-ds
```

- Delete the index template.

```
DELETE _index_template/metrics_template
```

Documentation

- Index templates
- Data streams
- Mapping types

Example 4: Defining a Data Stream with Specific Lifecycle Policies

Requirements

- Create an index template named `logs_index_template`.
- Create a data stream named `logs_my_app_production`.
- Configure the data lifecycle:
  - Data is hot for 3 minutes.
  - Data rolls to warm immediately after 3 minutes.
  - Data is warm for 5 minutes.
  - Data rolls to cold after 5 minutes.
  - Data is deleted 10 minutes after rolling to cold.

Steps

1. **Create the Index Template**:
   - Define an index template named `logs_index_template` that matches the data stream `logs_my_app_production`.

```
PUT _index_template/logs_index_template
{
  "index_patterns": ["logs_my_app_production*"],
  "data_stream": {}
}
```

2. Create the ILM Policy using the Elastic/Kibana UI{.unnumbered}
   1. Open the hamburger menu and click on **Management > Data > Index Life Cycle Policies**.
   2. Press **+ Create New Policy**.
   3. Enter the following:

- Policy name: **logs-policy**
- **Hot** phase:
  - Advanced Settings > Use Recommended Defaults (disable) > Maximum Age: 7 Days
- **Warm** phase (enable):
  - Move data into phase when: **3 minutes** old.
  - Leave **Delete data after this phase**.
- **Cold** phase:

- Move data into phase when: **5 minutes** old.
- Leave **Delete data after this phase**.

- **Delete** phase:
  - Move data into phase when: **10 minutes** old.

4. Press **Save Policy**.
5. **Management > Data > Index Life Cycle Policies > [plus sign]**
6. Add Policy "logs-policy" to Index Template > Index Template: logs_index_template > Add Policy

OR

2. **Create the ILM Policy**:
   - Define an Index Lifecycle Management (ILM) policy named `logs_index_policy` to manage the data lifecycle.

```
PUT _ilm/policy/logs_index_policy
{
  "policy": {
    "phases": {
      "hot": {
        "min_age": "0ms",
        "actions": {
          "rollover": {
            "max_age": "3m"
          }
        }
      },
      "warm": {
        "min_age": "3m",
        "actions": {
          "set_priority": {
            "priority": 50
          }
        }
      },
      "cold": {
        "min_age": "8m",
        "actions": {
          "set_priority": {
            "priority": 0
```

```
          }
        }
      },
      "delete": {
        "min_age": "18m",
        "actions": {
          "delete": {}
        }
      }
    }
  }
}
```

3. **Create the Data Stream**:
    - Creating the data stream is similar to creating an index using:

```
PUT logs_my_app_production
```

    - Create the data stream

```
PUT /_data_stream/logs_my_app_production
```

Test

1. **Index Sample Data**:
    - Index some sample documents into the data stream to ensure it is working correctly.

```
POST /logs_my_app_production/_doc
{
  "message": "This is a test log entry",
  "@timestamp": "2024-07-10T23:00:00Z"
}
```

2. **Verify ILM Policy**:
    - Check the status of the ILM policy to ensure it is being applied correctly.

```
GET /_ilm/explain/logs-policy
```

3. **Monitor Data Lifecycle**:
    - Monitor the data stream to ensure that documents transition through the hot, warm, cold, and delete phases as expected.

Considerations

- The `rollover` action in the hot phase ensures that the index rolls over after 3 minutes.
- The `set_priority` action in the warm and cold phases helps manage resource allocation.
- The `delete` action in the delete phase ensures that data is deleted 10 minutes after rolling to cold.

Clean-up (Optional)

- Delete the data stream and index template to clean up the resources.

```
DELETE /_data_stream/logs_my_app_production
DELETE /_index_template/logs_index_template
DELETE /_ilm/policy/logs-policy
```

Documentation

- Elasticsearch Data Streams
- Elasticsearch Index Templates
- Elasticsearch ILM Policies

2 Searching Data

2.1 Task: Write and execute a search query for terms and/or phrases in one or more fields of an index

The following section will have only one full example, but will show variations of **term** and **phrase** queries. Also, bear in mind that when they say **term** they may not mean the Elasticsearch use of the word, but rather the generic search use of the word. There are a lot of ways to execute a search in Elasticsearch. Don't get bogged down; focus on **term** and **phrase** searches for this section of the example.

Example 1: Write and execute a basic term and phrase search

Requirements

- Create an index
- Index some documents
- Execute a **term** query
- Execute a **phrase** query

Steps

1. **Open the Kibana Console** or use a REST client.
2. Index some documents which will create an index at the same time. The Elastic Console doesn't like properly formatted documents when calling `_bulk` so they need to be tightly packed.

```
POST /example_index/_bulk
{ "index": {} }
{ "title": "The quick brown fox", "text": "The quick brown fox jumps
 ↪  over the lazy dog." }
{ "index": {} }
{ "title": "Fast and curious", "text": "A fast and curious fox was seen
 ↪  leaping over a lazy dog." }
```

```
{ "index": {} }
{ "title": "A fox in action", "text": "In a remarkable display of agility, a quick fox effortlessly jumped over a dog." }
{ "index": {} }
{ "title": "Wildlife wonders", "text": "Observers were amazed as the quick brown fox jumped over the lazy dog." }
{ "index": {} }
{ "title": "Fox tales", "text": "The tale of the quick fox that jumped over the lazy dog has become a legend." }
```

3. Execute a `term` query

- Use the **GET** method to search for documents using 3 different term queries (there are 10 different ways currently. Refer to the Term-level Queries documentation for the full list).

```
GET example_index/_search
{
  "query": {
    "term": {
      "title": {
        "value": "quick"
      }
    }
  }
}
```

```
GET example_index/_search
{
  "query": {
    "terms": {
      "text": ["display", "amazed"]
    }
  }
}
```

4. Execute a `phrase` query

- returns 2 docs

```
GET /example_index/_search
{
  "query": {
```

```
    "match_phrase": {
      "text": "quick brown fox"
    }
  }
}
```

- returns 1 doc

```
GET /example_index/_search
{
  "query": {
    "match_phrase_prefix": {
      "text": "fast and curi"
    }
  }
}
```

- returns 1 doc

```
GET /example_index/_search
{
  "query": {
    "query_string": {
      "default_field": "text",
      "query": "\"fox jumps\""
    }
  }
}
```

Considerations

- The default **standard analyzer** (lowercasing, whitespace tokenization, basic normalization) is used.
- The `term` query is used for exact matches and is not analyzed, meaning it matches the exact term in the inverted index.
- The `match_phrase` query analyzes the input text and matches it as a phrase, making it useful for finding exact sequences of terms.

Test

1. Verify the various queries return the proper results.

Clean-up (optional)

- Delete the example index

```
DELETE example_index
```

Documentation

- Full Text Queries
- Match Phrase Query
- Match Phrase Prefix Query
- Query DSL
- Term-level Queries

Example 2: Boosting Document Score When an Additional Field Matches

Requirements

- Perform a search for `beverage OR bar`
- Boost the score of documents if the value `snack` exists in the `tags` field.

Steps

1. **Index Sample Documents Using `_bulk` Endpoint:**
   - Index documents with fields such as `name`, `description`, and `tags`.

```
POST /products/_bulk
{ "index": { "_id": "1" } }
{ "name": "Yoo-hoo Beverage", "description": "A delicious, chocolate-flavored drink.", "tags": ["beverage", "chocolate"] }
{ "index": { "_id": "2" } }
{ "name": "Apple iPhone 12", "description": "The latest iPhone model with advanced features.", "tags": ["electronics", "smartphone"] }
{ "index": { "_id": "3" } }
{ "name": "Choco-Lite Bar", "description": "A light and crispy chocolate snack bar.", "tags": ["snack", "chocolate"] }
{ "index": { "_id": "4" } }
{ "name": "Samsung Galaxy S21", "description": "A powerful smartphone with an impressive camera.", "tags": ["electronics", "smartphone"] }
{ "index": { "_id": "5" } }
{ "name": "Nike Air Max 270", "description": "Comfortable and stylish sneakers.", "tags": ["footwear", "sportswear"] }
```

2. **Perform the `query_string` Query with Boosting:**
   - Use a `query_string` query to create an OR condition within the query.
   - Use a `function_score` query to boost the score of documents where the `tags` field contains a specific value (e.g., `"chocolate"`).

```
GET /products/_search
{
  "query": {
    "function_score": {
      "query": {
        "query_string": {
          "query": "beverage OR bar"
```

```
        }
      },
      "functions": [
        {
          "filter": {
            "term": { "tags": "snack" }
          },
          "weight": 2
        }
      ],
      "boost_mode": "multiply"
    }
  }
}
```

Test

- Run the above search query.
- Run the following query (which is missing the `filter` function)

```
GET /products/_search
{
  "query": {
    "query_string": {
      "query": "beverage OR bar"
    }
  }
}
```

- Check the boosted output to ensure that documents containing `"snack"` in the `tags` field have a higher score, and that documents are matched based on the OR condition in the `query_string`.

Considerations

- The `query_string` query allows you to use a query syntax that includes operators such as `OR`, `AND`, and `NOT` to combine different search criteria.
- The `function_score` query is used to boost the score of documents based on specific conditions—in this case, whether the `tags` field contains the value `"snack"`.

- The `weight` parameter in the `function_score` query determines the amount by which the score is boosted, and the `boost_mode` of `"multiply"` multiplies the original score by the boost value.

Clean-up (optional)

- Delete the example index

```
DELETE products
```

Documentation

- Query String Query
- Function Score Query
- Term Query

2.2 Task: Write and execute a search query that is a Boolean combination of multiple queries and filters

Example 1: Creating a Boolean search for documents in a book index

Requirements

- Search for documents with a term in the "title", "description", and "category" field

Steps

1. **Open the Kibana Console** or use a REST client.
2. Index some documents which will create an index at the same time. The Elastic Console doesn't like properly formatted documents when calling **_bulk** so they need to be tightly packed.

```
POST /books/_bulk
{ "index": { "_id": "1" } }
{ "title": "To Kill a Mockingbird", "description": "A novel about the serious issues of rape and racial inequality.", "category": "Fiction" }
{ "index": { "_id": "2" } }
{ "title": "1984", "description": "A novel that delves into the dangers of totalitarianism.", "category": "Dystopian" }
{ "index": { "_id": "3" } }
{ "title": "The Great Gatsby", "description": "A critique of the American Dream.", "category": "Fiction" }
{ "index": { "_id": "4" } }
{ "title": "Moby Dick", "description": "The quest of Ahab to exact revenge on the whale Moby Dick.", "category": "Adventure" }
{ "index": { "_id": "5" } }
{ "title": "Pride and Prejudice", "description": "A romantic novel that also critiques the British landed gentry at the end of the 18th century.", "category": "Romance" }
```

3. Create a boolean search query. The order in which the various clauses are added don't matter to the final result.

```
GET books/_search
{
  "query": {
```

```
    "bool": {}
  }
}
```

4. Add a `must` query for the description field. This will return 4 documents.

```
GET books/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "terms": {
            "description": [
              "novel",
              "dream",
              "critique"
            ]
          }
        }
      ]
    }
  }
}
```

5. Add a `filter` query for the category field. This will return 2 documents.

```
GET books/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "terms": {
            "description": [
              "novel",
              "dream",
              "critique"
            ]
          }
        }
      ],
      "filter": [
```

```
        {
          "term": {
            "category": "fiction"
          }
        }
      ]
    }
  }
}
```

6. Add a `must_not` filter for the title field. This will return 1 document.

```
GET books/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "terms": {
            "description": [
              "novel",
              "dream",
              "critique"
            ]
          }
        }
      ],
      "filter": [
        {
          "term": {
            "category": "fiction"
          }
        }
      ],
      "must_not": [
        {
          "term": {
            "title": {
              "value": "gatsby"
            }
          }
        }
```

```
      ]
    }
  }
}
```

Considerations

- The `bool` query allows for combining multiple queries and filters with Boolean logic.
- The `must`, `must_not`, and `filter` clauses ensure that all searches and filters must match for a document to be returned.

Test

1. Verify that the search query returns documents with the term "novel", "dream", and "critique" in the `description` field. Why are there no documents with the term "critique"?

Clean-up (optional)

- Delete the index

```
DELETE books
```

Documentation

- Elasticsearch Boolean Query
- Elasticsearch Match Query
- Elasticsearch Range Query
- Elasticsearch Term Query

Example 2: Creating a Boolean search for finding products within a specific price range and excluding discontinued items

Requirements

- Find all documents where the `name` field exists (`name: \*`) and the `price` field falls within a specified range.
- Additionally, filter out any documents where the `discontinued` field is set to `true`.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Index some documents which will create an index at the same time. The Elastic Console doesn't like properly formatted documents when calling **_bulk** so they need to be tightly packed.

```
POST /products/_bulk
{"index":{"_id":1}}
{"name":"Coffee Maker","price":49.99,"discontinued":false}
{"index":{"_id":2}}
{"name":"Gaming Laptop","price":1299.99,"discontinued":false}
{"index":{"_id":3}}
{"name":"Wireless Headphones","price":79.99,"discontinued":true}
{"index":{"_id":4}}
{"name":"Smartwatch","price":249.99,"discontinued":false}
```

3. Construct the first search query (the `name` field exists and the `price` field falls within a specified range)

```
GET products/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "exists": {
            "field": "name"
          }
        },
        {
          "range": {
            "price": {
```

```
              "gte": 70,
              "lte": 500
            }
          }
        }
      ]
    }
  }
}
```

4. Construct the second search query (same as above, but check if `discontinued` is set to `true`)

```
GET products/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "exists": {
            "field": "name"
          }
        },
        {
          "range": {
            "price": {
              "gte": 70,
              "lte": 500
            }
          }
        }
      ],
      "must_not": [
        {
          "term": {
            "discontinued": {
              "value": "true"
            }
          }
        }
      ]
    }
```

```
  }
}
```

Explanation

- Similar to the previous example, the `bool` query combines multiple conditions.
- The `must` clause specifies documents that must match all conditions within it.
- The `range` query ensures the price field is between $70 (inclusive) and $500 (inclusive).
- The `must_not` clause excludes documents that match the specified criteria.
- The `term` query filters out documents where `discontinued` is set to true.

Test

1. Run the search query and verify the results only include documents for products with:
    - A `price` between $70 and $500 (inclusive).
    - `discontinued` set to `true` (not discontinued).

This should return a single document with an ID of 4 (Smartwatch) based on the sample data.

Considerations

- The chosen price range (gte: 70, lte: 500) can be adjusted based on your specific needs.
- You can modify the `match` query for `name` to use more specific criteria if needed.

Clean-up (optional)

- Delete the index

```
DELETE products
```

Documentation

- Elasticsearch Boolean Query
- Elasticsearch Match Query
- Elasticsearch Range Query
- Elasticsearch Term Query

Example 3: Creating a Boolean search for e-commerce products

Requirements

- Search for products that belong to the "Electronics" category.
- The product name should contain the term "phone".
- Exclude products with a price greater than 500.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create an index.

```
PUT products
{
  "mappings": {
    "properties": {
      "name" : {
        "type": "text"
      },
      "category" : {
        "type": "text"
      },
      "price" : {
        "type": "float"
      }
    }
  }
}
```

3. Index some documents which will create an index at the same time. The Elastic Console doesn't like properly formatted documents when calling **_bulk** so they need to be tightly packed.

```
POST /products/_bulk
{"index": { "_id": 1 } }
{ "name": "Smartphone X", "category": "Electronics", "price": 399.99 }
{"index": { "_id": 2 } }
{ "name": "Laptop Y", "category": "Electronics", "price": 799.99 }
{"index": { "_id": 3 } }
{ "name": "Headphones Z", "category": "Electronics", "price": 99.99 }
```

```
{"index": { "_id": 4 } }
{ "name": "Gaming Console", "category": "Electronics", "price": 299.99 }
```

4. Create a `term` query that only matches the category "electronics". This returns all 4 documents.

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

5. Create another query using `wildcard` to return docs that includes "phone". This returns only 2 documents.

```
GET products/_search
{
  "query": {
    "wildcard": {
      "name": {
        "value": "*phone*"
      }
    }
  }
}
```

6. Create another query using `range` that returns docs with any price less than $500. This returns 3 documents.

```
GET products/_search
{
  "query": {
    "range": {
      "price": {
        "lt": 500
      }
    }
  }
```

```
}
```

7. Combine the above into one `bool` query with a single `must` that contains the three queries. This will return the 2 matching documents.

```
GET products/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "term": {
            "category": {
              "value": "electronics"
            }
          }
        },
        {
          "wildcard": {
            "name": {
              "value": "*phone*"
            }
          }
        },
        {
          "range": {
            "price": {
              "lt": 500
            }
          }
        }
      ]
    }
  }
}
```

Test

1. The search results should include the following documents:
   - Smartphone X
   - Headphones Z

Considerations

- The `term` query is used for matches on the `category` field.
- The `wildcard` query is used for matches on the `name` field.
- The `range` query is used to filter out documents based on `price`.
- The `bool.must` query combines these conditions using the specified occurrence types.

Clean-up (optional)

- Delete the index

```
DELETE products
```

Documentation

- Boolean Query
- Match Query
- Range Query
- Term Query
- Wildcard Query

Example 4: Creating a Boolean search for e-commerce products

Requirements

- Create an index named "products".
- Create at least 4 documents with varying categories, prices, ratings, and brands.
- Create a boolean query
  - Use the `must`:
    - return just electronics
    - products more than $500
  - Use `must_not`:
    - rating less than 4
  - Use `filter`:
    - only Apple products

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the "products" index

```
PUT products
{
  "mappings": {
    "properties": {
      "brand": {
        "type": "text"
      },
      "category": {
        "type": "keyword"
      },
      "name": {
        "type": "text"
      },
      "price": {
        "type": "long"
      },
      "rating": {
        "type": "float"
      }
    }
```

```
  }
}
```

3. Add some sample documents using the `_bulk` endpoint.

```
POST /products/_bulk
{"index":{"_id":1}}
{"name":"Laptop","category":"Electronics","price":1200,"rating":4.5,"brand":"Apple"}
{"index":{"_id":2}}
{"name":"Smartphone","category":"Electronics","price":800,"rating":4.2,"brand":"Samsung"
{"index":{"_id":3}}
{"name":"Sofa","category":"Furniture","price":1000,"rating":3.8,"brand":"IKEA"}
{"index":{"_id":4}}
{"name":"Headphones","category":"Electronics","price":150,"rating":2.5,"brand":"Sony"}
{"index":{"_id":5}}
{"name":"Dining
 ↪ Table","category":"Furniture","price":600,"rating":4.1,"brand":"Ashley"}
```

4. Create a `term` query that only matches the category "electronics". This returns 3 documents.

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

5. Create a `range` query to return products whose price is greater than $500. This should return 4 documents (why?).

```
GET products/_search
{
  "query": {
    "range": {
      "price": {
        "gte": 500
      }
    }
```

```
  }
}
```

6. Create another `range` query to return products with a rating less than 4. This will return 2 documents.

```
GET products/_search
{
  "query": {
    "range": {
      "rating": {
        "lt": 4
      }
    }
  }
}
```

7. Create another `term` query to return only Apple branded products. This will return 2 documents.

```
GET products/_search
{
  "query": {
    "term": {
      "brand": {
        "value": "apple"
      }
    }
  }
}
```

8. Assemble the `bool` query by placing each query in their appropriate `must`, `must_not` and `filter` node.

```
GET products/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "term": {
            "category": {
              "value": "electronics"
```

```
            }
          }
        },
        {
          "range": {
            "price": {
              "gte": 500
            }
          }
        }
      ],
      "must_not": [
        {
          "range": {
            "rating": {
              "lt": 4
            }
          }
        }
      ],
      "filter": [
        {
          "term": {
            "brand": {
              "value": "apple"
            }
          }
        }
      ]
    }
  }
}
```

Test

- Check the response from the search query to ensure that it returns the expected documents
  - products in the “Electronics” category
  - a price greater than $500
  - excluding products with a rating less than 4
  - from the brand “Apple”

Considerations

- The filter clause is used to include only documents with the brand “Apple”.

Clean-up (optional)

- Delete the index

```
DELETE products
```

Documentation

- Elasticsearch Boolean Query
- Elasticsearch Term Query
- Elasticsearch Range Query

2.3 Task: Create an asynchronous search

Asynchronous search uses the same parameters as regular search with a few extra features listed here. For example, in the solution below the documentation for the size option is here. There is only one example here as you can look up the other options as needed during the exam.

Example 1: Executing an asynchronous search on a large log index

Requirements

- An Elasticsearch index named "logs" with a large number of documents (e.g., millions of log entries).
- Perform a search on the "logs" index that may take a long time to complete due to the size of the index.
- Retrieve the search results asynchronously without blocking the client.

Steps

1. **Open the Kibana Console** or use a REST client.
2. If you were submitting a normal/synchronous search to an index called `logs` your request would look something like this:

```
POST /logs/_search
{
  "query": {
    "match_all": {}
  },
  "size": 10000
}
```

3. To turn your request into an asynchronous search request turn `_search` to `_async_search`

```
POST /logs/_async_search
{
  "query": {
    "match_all": {}
  },
  "size": 10000
}
```

This request will return an id and a response object containing partial results if available.

4. Check the status of the asynchronous search using the id.

```
GET /_async_search/status/{id}
```

5. Retrieve the search results using the id.

```
GET /_async_search/{id}
```

Test

1. Index a large number of sample log documents or use an index with a large number of documents.
2. Execute the asynchronous search request and store the returned **id**.
3. Periodically check the status of the search using the id and the `/_async_search/status/{id}` endpoint.

```
GET /_async_search/status/{id}
```

4. Once the search is complete, retrieve the final results using the **id** and the `/_async_search/{id}` endpoint.

```
GET /_async_search/{id}
```

Considerations

- The `_async_search` endpoint is used to submit an asynchronous search request.
- The **id** returned by the initial request is used to check the status and retrieve the final results.
- Asynchronous search is useful for long-running searches on large datasets, as it doesn't block the client while the search is being processed.

Clean-up (optional)

- If you created an index (for example, `logs`) for this example you might want to delete it.

```
DELETE logs
```

Documentation

- Async Search API
- Submitting Async Search
- Status Check Async Search
- Retrieving Async Search Results

2.4 Task: Write and execute metric and bucket aggregations

Example 1: Creating Metric and Bucket Aggregations for Product Prices

Requirements

- Create an index called `product_prices`.
- Index at least four documents using the `_bulk` endpoint.
- Execute metric and bucket aggregations in a single
  - bucket the `category` field
  - calculate the average price per bucket
  - find the maximum price per bucket
  - find the minimum price per bucket

Steps

1. **Open the Kibana Console** or use a REST client.

Ensure you have access to Kibana or any REST client to execute the following requests.

2. Create an index with the following schema (needed for the aggregations to work properly).

```
PUT product_prices
{
  "mappings": {
    "properties": {
      "product": {
        "type": "text"
      },
      "category": {
        "type": "keyword"
      },
      "price": {
        "type": "double"
      }
    }
  }
}
```

3. Index documents.

```
POST /product_prices/_bulk
{ "index": { "_id": "1" } }
{ "product": "Elasticsearch Guide", "category": "Books", "price": 29.99 }
{ "index": { "_id": "2" } }
{ "product": "Advanced Elasticsearch", "category": "Books", "price": 39.99 }
{ "index": { "_id": "3" } }
{ "product": "Elasticsearch T-shirt", "category": "Apparel", "price": 19.99 }
{ "index": { "_id": "4" } }
{ "product": "Elasticsearch Mug", "category": "Apparel", "price": 12.99 }
```

4. Execute a simple aggregation (should return 2 buckets).

```
GET product_prices/_search
{
  "size": 0,
  "aggs": {
    "category_buckets": {
      "terms": {
        "field": "category"
      }
    }
  }
}
```

5. Add and execute a single sub-aggregation to determine the average price per category (bucket).

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

```
          }
        }
      }
    }
  }
}
```

6. Add min and max sub-aggregations and execute the query.

```
GET product_prices/_search
{
  "size": 0,
  "aggs": {
    "category_buckets": {
      "terms": {
        "field": "category"
      },
      "aggs": {
        "avg_price": {
          "avg": {
            "field": "price"
          }
        },
        "min_price" : {
          "min": {
            "field": "price"
          }
        },
        "max_price": {
          "max": {
            "field": "price"
          }
        }
      }
    }
  }
}
```

Test

1. Verify the index creation.

```
GET /product_prices
```

2. Verify the documents have been indexed.

```
GET /product_prices/_search
```

3. Execute the aggregation query and verify the results.

```
{
  ...
  "aggregations": {
    "category_buckets": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": "Apparel",
          "doc_count": 2,
          "avg_price": {
            "value": 16.49
          },
          "min_price": {
            "value": 12.99
          },
          "max_price": {
            "value": 19.99
          }
        },
        {
          "key": "Books",
          "doc_count": 2,
          "avg_price": {
            "value": 34.99
          },
          "min_price": {
            "value": 29.99
          },
          "max_price": {
            "value": 39.99
          }
        }
      ]
    }
```

```
  }
}
```

Considerations

- The category field must be of type `keyword`.
- The `terms` aggregation creates buckets for each unique category.
- The `avg`, `min`, and `max` sub-aggregations calculate the average, minimum, and maximum prices within each category bucket.
- Setting size to 0 ensures that only aggregation results are returned, not individual documents.

Clean-up (optional)

- Delete the index.

```
DELETE product_prices
```

Documentation

- Aggregations
- Terms Aggregation
- Avg Aggregation
- Max Aggregation
- Min Aggregation

Example 2: Creating Metric and Bucket Aggregations for Website Traffic

Requirements

- Create a new index with four documents representing website traffic data.
- Aggregate the following:
  - Group traffic by country.
  - Calculate the total page views.
  - Calculate the average page views per country.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create a new index.

```
PUT traffic
{
  "mappings": {
    "properties": {
      "country": {
        "type": "keyword"
      },
      "page_views": {
        "type": "long"
      }
    }
  }
}
```

3. Add four documents representing website traffic data.

```
POST /traffic/_bulk
{"index":{}}
{"country":"USA","page_views":100}
{"index":{}}
{"country":"USA","page_views":200}
{"index":{}}
{"country":"Canada","page_views":50}
{"index":{}}
{"country":"Canada","page_views":75}
```

4. Execute the bucket aggregation for `country` (should return 2 buckets).

```
GET traffic/_search
{
  "size": 0,
  "aggs": {
    "country_bucket": {
      "terms": {
        "field": "country"
      }
    }
  }
}
```

5. Add the `sum` aggregation for total `page_views` (should return 1 aggregation).

```
GET traffic/_search
{
  "size": 0,
  "aggs": {
    "country_bucket": {
      "terms": {
        "field": "country"
      }
    },
    "total_page_views": {
      "sum": {
        "field": "page_views"
      }
    }
  }
}
```

6. Add a sub-aggregation for average `page_views` per `country` (should appear in 2 buckets).

```
GET traffic/_search
{
  "size": 0,
  "aggs": {
    "country_bucket": {
      "terms": {
        "field": "country"
      },
      "aggs": {
```

```
        "avg_page_views": {
          "avg": {
            "field": "page_views"
          }
        }
      }
    },
    "total_page_views": {
      "sum": {
        "field": "page_views"
      }
    }
  }
}
```

Test

1. Verify the index creation.

```
GET /traffic
```

2. Verify the documents have been indexed.

```
GET /traffic/_search
```

3. Verify that the total page views are calculated correctly (should be 425).

```
GET /traffic/_search
{
  "aggs": {
    "total_page_views": {
      "sum": {
        "field": "page_views"
      }
    }
  }
}
```

4. Verify that the traffic is grouped correctly by country and average page views are calculated.

```
GET /traffic/_search
{
  "aggs": {
    "traffic_by_country": {
      "terms": {
        "field": "country"
      },
      "aggs": {
        "avg_page_views": {
          "avg": {
            "field": "page_views"
          }
        }
      }
    }
  }
}
```

Response:

```
{
  ...
  "aggregations": {
    "country_bucket": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": "Canada",
          "doc_count": 2,
          "avg_page_views": {
            "value": 62.5
          }
        },
        {
          "key": "USA",
          "doc_count": 2,
          "avg_page_views": {
            "value": 150
          }
        }
      ]
```

```
    },
    "total_page_views": {
      "value": 425
    }
  }
}
```

Considerations

- The `country` field must be of type keyword.
- The `terms` bucket aggregation is used to group traffic by country.
- The `sum` metric aggregation is used to calculate the total page views.
- The `avg` metric aggregation is used to calculate the average page views per country.

Clean-up (optional)

- Delete the index.

```
DELETE traffic
```

Documentation

- Aggregations
- Metric Aggregations
- Bucket Aggregations
- Terms Aggregation

Example 3: Creating Metric and Bucket Aggregations for Analyzing Employee Salaries

Requirements

- An Elasticsearch index named `employees` with documents containing fields `name`, `department`, `position`, `salary`, `hire_date`.
- Calculate the average salary across all employees.
- Group the employees by department
- Calculate the maximum salary for each department.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create an index with the proper mapping for the department as we want to bucket by it.

```
PUT employees
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "department": {
        "type": "keyword"
      },
      "position": {
        "type": "text"
      },
      "salary": {
        "type": "integer"
      },
      "hire_date": {
        "type": "date"
      }
    }
  }
}
```

3. Index sample employee documents using the /_bulk endpoint.

```
POST /employees/_bulk
{"index":{"_id":1}}
{"name":"John Doe", "department":"Engineering", "position":"Software Engineer", "salary":80000, "hire_date":"2018-01-15"}
{"index":{"_id":2}}
{"name":"Jane Smith", "department":"Engineering", "position":"DevOps Engineer", "salary":75000, "hire_date":"2020-03-01"}
{"index":{"_id":3}}
{"name":"Bob Johnson", "department":"Sales", "position":"Sales Manager", "salary":90000, "hire_date":"2016-06-01"}
{"index":{"_id":4}}
{"name":"Alice Williams", "department":"Sales", "position":"Sales Representative", "salary":65000, "hire_date":"2019-09-15"}
```

4. Calculate the average salary of all employees

```
GET employees/_search
{
  "size": 0,
  "aggs": {
    "avg_salary_all_emps": {
      "avg": {
        "field": "salary"
      }
    }
  }
}
```

5. Add grouping the employees by department

```
GET employees/_search
{
  "size": 0,
  "aggs": {
    "avg_salary_all_emps": {
      "avg": {
        "field": "salary"
      }
    },
    "employees_by_department" : {
      "terms": {
        "field": "department"
      }
```

```
    }
  }
}
```

6. Add calculating the highest salary of all employees by department

```
GET employees/_search
{
  "size": 0,
  "aggs": {
    "avg_salary_all_emps": {
      "avg": {
        "field": "salary"
      }
    },
    "employees_by_department": {
      "terms": {
        "field": "department"
      },
      "aggs": {
        "max_salary_by_department": {
          "max": {
            "field": "salary"
          }
        }
      }
    }
  }
}
```

Test

1. Verify the index creation.

```
GET /employees
```

2. Verify the documents have been indexed.

```
GET /employees/_search
```

3. Execute the aggregation query, and it should return the following:

```
{
  ...
  "aggregations": {
    "avg_salary_all_emps": {
      "value": 77500
    },
    "employees_by_department": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": "Engineering",
          "doc_count": 2,
          "max_salary_by_department": {
            "value": 80000
          }
        },
        {
          "key": "Sales",
          "doc_count": 2,
          "max_salary_by_department": {
            "value": 90000
          }
        }
      ]
    }
  }
}
```

Considerations

- The `department` field must be of type keyword.
- The `size` parameter is set to 0 to exclude hit documents from the response.
- The `avg_salary_all_emps` metric aggregation calculates the average of the `salary` field across all documents.
- The `employees_by_department` bucket aggregation groups the documents by the department field.
- The `max_salary_by_department` sub-aggregation calculates the maximum value of the salary field for each department.

Clean-up (optional)

- Delete the index.

```
DELETE employees
```

Documentation

- Elasticsearch Aggregations
- Metric Aggregations
- Bucket Aggregations
- Terms Aggregation

2.5 Task: Write and execute aggregations that contain subaggregations

Example 1: Creating aggregations and sub-aggregations for Product Categories and Prices

Requirements

- Create aggregations
  - by category
  - sub-aggregation of average price by category
    - price ranges: $0 to $20, $20-$40, $40 and up

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create an index.

```
PUT /product_index
{
  "mappings": {
    "properties": {
      "product": {
        "type": "text"
      },
      "category": {
        "type": "keyword"
      },
      "price": {
        "type": "double"
      }
    }
  }
}
```

3. Index some sample documents.

```
POST /product_index/_bulk
{ "index": { "_id": "1" } }
{ "product": "Elasticsearch Guide", "category": "Books", "price": 29.99
 ↪  }
```

```
{ "index": { "_id": "2" } }
{ "product": "Advanced Elasticsearch", "category": "Books", "price": 39.99 }
{ "index": { "_id": "3" } }
{ "product": "Elasticsearch T-shirt", "category": "Apparel", "price": 19.99 }
{ "index": { "_id": "4" } }
{ "product": "Elasticsearch Mug", "category": "Apparel", "price": 12.99 }
```

4. Create an aggregation by category.

```
GET product_index/_search
{
  "size": 0,
  "aggs": {
    "category_buckets": {
      "terms": {
        "field": "category"
      }
    }
  }
}
```

5. Create a sub-aggregations of average price.

```
GET product_index/_search
{
  "size": 0,
  "aggs": {
    "category_buckets": {
      "terms": {
        "field": "category"
      },
      "aggs": {
        "average_price": {
          "avg": {
            "field": "price"
          }
        }
      }
    }
  }
```

```
}
```

6. Create a sub-aggregations of price ranges ($0-$20, $10-$40, $40 and up).

```
GET product_index/_search
{
  "size": 0,
  "aggs": {
    "category_buckets": {
      "terms": {
        "field": "category"
      },
      "aggs": {
        "average_price": {
          "avg": {
            "field": "price"
          }
        },
        "price_ranges" : {
          "range": {
            "field": "price",
            "ranges": [
              {
                "to": 20
              },
              {
                "from": 20,
                "to": 40
              },
              {
                "from": 40
              }
            ]
          }
        }
      }
    }
  }
}
```

Test

1. Verify the index creation and mappings.

```
GET /product_index
```

2. Verify the test documents are in the index.

```
GET /product_index/_search
```

3. Execute the aggregation query and confirm the results.

```
{
  ...
  "aggregations": {
    "category_buckets": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": "Apparel",
          "doc_count": 2,
          "average_price": {
            "value": 16.49
          },
          "price_ranges": {
            "buckets": [
              {
                "key": "*-20.0",
                "to": 20,
                "doc_count": 2
              },
              {
                "key": "20.0-40.0",
                "from": 20,
                "to": 40,
                "doc_count": 0
              },
              {
                "key": "40.0-*",
                "from": 40,
                "doc_count": 0
              }
            ]
```

```
          }
        },
        {
          "key": "Books",
          "doc_count": 2,
          "average_price": {
            "value": 34.99
          },
          "price_ranges": {
            "buckets": [
              {
                "key": "*-20.0",
                "to": 20,
                "doc_count": 0
              },
              {
                "key": "20.0-40.0",
                "from": 20,
                "to": 40,
                "doc_count": 2
              },
              {
                "key": "40.0-*",
                "from": 40,
                "doc_count": 0
              }
            ]
          }
        }
      ]
    }
  }
}
```

Considerations

- Setting `size`: 0 ensures the search doesn't return any documents, focusing solely on the aggregations.
- The `category` field must be of type keyword.
- The `terms` aggregation creates buckets for each unique category.
- The `avg` sub-aggregation calculates the average price within each category bucket.

- The `range` sub-aggregation divides the prices into specified ranges within each category bucket.

Clean-up (optional)

- Delete the index.

```
DELETE product_index
```

Documentation

- Aggregations
- Avg Aggregation
- Range Aggregation
- Terms Aggregation

Example 2: Creating aggregations and sub-aggregations for Employee Data Analysis

Requirements

- Use the `terms` aggregation to group employees by department.
- Use the `avg` sub-aggregation to calculate the average salary per department.
- Use the `filters` sub-aggregation to group employees by job_title.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create a new index called `employees`.

```
PUT employees
{
  "mappings": {
    "properties": {
      "department": {
        "type": "keyword"
      },
      "salary": {
        "type": "integer"
      },
      "job_title": {
        "type": "keyword"
      }
    }
  }
}
```

3. Insert four documents representing employee data.

```
POST /employees/_bulk
{"index":{}}
{"department":"Sales","salary":100000,"job_title":"Manager"}
{"index":{}}
{"department":"Sales","salary":80000,"job_title":"Representative"}
{"index":{}}
{"department":"Marketing","salary":120000,"job_title":"Manager"}
{"index":{}}
{"department":"Marketing","salary":90000,"job_title":"Coordinator"}
```

4. Execute an aggregation by department.

```
GET employees/_search
{
  "size": 0,
  "aggs": {
    "employees_by_department": {
      "terms": {
        "field": "department"
      }
    }
  }
}
```

5. Add the sub-aggregations for average salary by department.

```
GET employees/_search
{
  "size": 0,
  "aggs": {
    "employees_by_department": {
      "terms": {
        "field": "department"
      },
      "aggs": {
        "avg_salary_by_department": {
          "avg": {
            "field": "salary"
          }
        }
      }
    }
  }
}
```

6. Add a filters sub-aggregation for each `job_title`.

```
GET employees/_search
{
  "size": 0,
  "aggs": {
    "employees_by_department": {
      "terms": {
        "field": "department"
```

```
      },
      "aggs": {
        "avg_salary_by_department": {
          "avg": {
            "field": "salary"
          }
        },
        "employees_by_title": {
          "filters": {
            "filters": {
              "Managers": {
                "term": {
                  "job_title": "Manager"
                }
              },
              "Representative" : {
                "term": {
                  "job_title": "Representative"
                }
              },
              "Coordinator" : {
                "term": {
                  "job_title": "Coordinator"
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Test

1. Verify the index creation and mappings.

```
GET /employees
```

2. Verify the test documents are in the index.

```
GET /employees/_search
```

3. Verify that the employees are grouped correctly by department and job title and that the average salary is calculated correctly for each department.

```
{
  ...
  "aggregations": {
    "employees_by_department": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": "Marketing",
          "doc_count": 2,
          "avg_salary_by_department": {
            "value": 105000
          },
          "employees_by_title": {
            "buckets": {
              "Coordinator": {
                "doc_count": 1
              },
              "Managers": {
                "doc_count": 1
              },
              "Representative": {
                "doc_count": 0
              }
            }
          }
        },
        {
          "key": "Sales",
          "doc_count": 2,
          "avg_salary_by_department": {
            "value": 90000
          },
          "employees_by_title": {
            "buckets": {
              "Coordinator": {
                "doc_count": 0
```

```
                },
                "Managers": {
                  "doc_count": 1
                },
                "Representative": {
                  "doc_count": 1
                }
              }
            }
          }
        ]
      }
    }
}
```

Considerations

- The `department` field must be of type keyword.
- Setting `size` to 0 ensures the search doesn't return any documents, focusing solely on the aggregations.
- The `terms` aggregation is used to group employees by department.
- The `avg` sub-aggregation is used to calculate the average salary per department.
- The `filters` sub-aggregation is used to group employees by `job_title`.

Clean-up (optional)

- Delete the index.

```
DELETE employees
```

Documentation

- Aggregations
- Avg Aggregation
- Filters Aggregation
- Range Aggregation
- Terms Aggregation

Example 3: Creating aggregations and sub-aggregations for application logs by Hour and Log Level

Requirements

- Analyze application logs stored in an Elasticsearch index named `app-logs`.
- Use a `date_histogram` aggregation to group logs by the hour.
- Within each hour bucket, create a sub-aggregation to group logs by their severity level (`log_level`).

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create a new index called app-logs.

```
PUT app-logs
{
  "mappings": {
    "properties": {
      "@timestamp": {
        "type": "date"
      },
      "log_level": {
        "type": "keyword"
      },
      "message": {
        "type": "text"
      }
    }
  }
}
```

3. Insert sample data.

```
POST /app-logs/_bulk
{"index":{},"_id":"1"}
{"@timestamp":"2024-05-24T10:30:00","log_level":"INFO","message":"Application started successfully."}
{"index":{},"_id":"2"}
{"@timestamp":"2024-05-24T11:15:00","log_level":"WARNING","message":"Potential memory leak detected."}
{"index":{},"_id":"3"}
```

```
{"@timestamp":"2024-05-24T12:00:00","log_level":"ERROR","message":"Database connection failed."}
{"index":{},"_id":"4"}
{"@timestamp":"2024-05-24T10:45:00","log_level":"DEBUG","message":"Processing user request."}
```

4. Use a `date_histogram` aggregation to group logs by the hour.

```
GET app-logs/_search
{
  "size": 0,
  "aggs": {
    "logs_by_the_hour": {
      "date_histogram": {
        "field": "@timestamp",
        "fixed_interval": "1h"
      }
    }
  }
}
```

5. Within each hour bucket, create a sub-aggregation to group logs by their severity level (log_level).

```
GET app-logs/_search
{
  "size": 0,
  "aggs": {
    "logs_by_the_hour": {
      "date_histogram": {
        "field": "@timestamp",
        "fixed_interval": "1h"
      },
      "aggs": {
        "log_severity": {
          "terms": {
            "field": "log_level"
          }
        }
      }
    }
  }
}
```

Test

1. Verify the index creation and mappings.

```
GET /app-logs
```

2. Verify the test documents are in the index.

```
GET /app-logs/_search
```

3. Run the search query and examine the response.

```
{
  ...
  "aggregations": {
    "logs_by_the_hour": {
      "buckets": [
        {
          "key_as_string": "2024-05-24T10:00:00.000Z",
          "key": 1716544800000,
          "doc_count": 2,
          "log_severity": {
            "doc_count_error_upper_bound": 0,
            "sum_other_doc_count": 0,
            "buckets": [
              {
                "key": "DEBUG",
                "doc_count": 1
              },
              {
                "key": "INFO",
                "doc_count": 1
              }
            ]
          }
        },
        {
          "key_as_string": "2024-05-24T11:00:00.000Z",
          "key": 1716548400000,
          "doc_count": 1,
          "log_severity": {
            "doc_count_error_upper_bound": 0,
            "sum_other_doc_count": 0,
            "buckets": [
```

```
            {
              "key": "WARNING",
              "doc_count": 1
            }
          ]
        }
      },
      {
        "key_as_string": "2024-05-24T12:00:00.000Z",
        "key": 1716552000000,
        "doc_count": 1,
        "log_severity": {
          "doc_count_error_upper_bound": 0,
          "sum_other_doc_count": 0,
          "buckets": [
            {
              "key": "ERROR",
              "doc_count": 1
            }
          ]
        }
      }
    ]
  }
 }
}
```

Considerations

- Setting `size` to 0 ensures the search doesn't return any documents, focusing solely on the aggregations.
- The `date_histogram` aggregation groups documents based on the `@timestamp` field with an interval of one hour.
- The nested `terms` aggregation within the `logs_by_hour` aggregation counts the occurrences of each unique `log_level` within each hour bucket.

Clean-up (optional)

- Delete the index.

```
DELETE app-logs
```

Documentation

- Bucket Aggregations
- Date Histogram Aggregation
- Terms Aggregation

Example 4: Finding the Stock with the Highest Daily Volume of the Month

This is taken from a webinar by Elastic to show a sample question and answer to the Certified Engineer Exam. Their answer was wrong and didn't need aggregations.

Requirements

- Create a query to find the stock with the highest daily volume for the current month.

Steps

(4) **Open the Kibana Console** or use a REST client.

(5) Index sample data:

- Use the `_bulk` endpoint to index sample stock data.
- Ensure the data includes fields for `stock_name`, `date`, and `volume`.

```
POST _bulk
{ "index": { "_index": "stocks", "_id": "1" } }
{ "stock_name": "AAPL", "date": "2024-07-01", "volume": 1000000 }
{ "index": { "_index": "stocks", "_id": "2" } }
{ "stock_name": "AAPL", "date": "2024-07-02", "volume": 1500000 }
{ "index": { "_index": "stocks", "_id": "3" } }
{ "stock_name": "GOOGL", "date": "2024-07-01", "volume": 2000000 }
{ "index": { "_index": "stocks", "_id": "4" } }
{ "stock_name": "GOOGL", "date": "2024-07-02", "volume": 2500000 }
{ "index": { "_index": "stocks", "_id": "5" } }
{ "stock_name": "MSFT", "date": "2024-07-01", "volume": 3000000 }
{ "index": { "_index": "stocks", "_id": "6" } }
{ "stock_name": "MSFT", "date": "2024-07-02", "volume": 3500000 }
{ "index": { "_index": "stocks", "_id": "7" } }
{ "stock_name": "TSLA", "date": "2024-07-01", "volume": 4000000 }
{ "index": { "_index": "stocks", "_id": "8" } }
{ "stock_name": "TSLA", "date": "2024-07-02", "volume": 4500000 }
{ "index": { "_index": "stocks", "_id": "9" } }
{ "stock_name": "AMZN", "date": "2024-07-01", "volume": 5000000 }
{ "index": { "_index": "stocks", "_id": "10" } }
{ "stock_name": "AMZN", "date": "2024-07-02", "volume": 5500000 }
```

(6) Create the query. The `stocks` in the index are all from July, but you want just the stocks for the latest month. Update the above dates so the query will work for you.

```
GET stocks/_search
{
  "size": 1,
  "query": {
    "range": {
      "date": {
        "gte": "now/M",
        "lte": "now"
      }
    }
  }
}
```

(7) The results of the `query` should be all the stocks from a given month. Now sort those stocks by their volume and display the top pick.

```
GET stocks/_search
{
  "size": 1,
  "query": {
    "range": {
      "date": {
        "gte": "now/M",
        "lte": "now"
      }
    }
  },
  "sort": [
    {
      "volume": {
        "order": "desc"
      }
    }
  ]
}
```

Test

1. Verify the index creation and mappings.

```
GET /stocks
```

2. Verify the test documents are in the index.

```
GET /stocks/_search
```

3. Run the query and confirm that the stock with the highest daily volume of the month is displayed.


```
{
  ...
    "hits": [
      {
        "_index": "stocks",
        "_id": "10",
        "_score": null,
        "_source": {
          "stock_name": "AMZN",
          "date": "2024-07-02",
          "volume": 5500000
        },
        "sort": [
          5500000
        ]
      }
    ]
  }
}
```


Considerations

- The `range` clause returned the stocks for the current month
- The `sort` clause brought the highest volume of any stock to the top and `size` of 1 displayed that one record

Clean-up (Optional)

- Delete the `stocks` index to clean up the data:

```
DELETE /stocks
```

Documentation

- Elasticsearch Bulk API
- Elasticsearch Date Histogram Aggregation
- Elasticsearch Max Aggregation
- Elasticsearch Top Hits Aggregation

Example 5: Aggregating Sales Data by Month with Sub-Aggregation of Total Sales Value

Requirements

- Aggregate e-commerce sales data by month, creating at least 12 date buckets.
- Perform a sub-aggregation to calculate the total sales value within each month.

Steps

1. **Index Sample Sales Documents Using `_bulk` Endpoint:**

```
POST /sales_data/_bulk
{ "index": { "_id": "1" } }
{ "order_date": "2023-01-15", "product": "Yoo-hoo Beverage", "quantity": 10, "price": 1.99 }
{ "index": { "_id": "2" } }
{ "order_date": "2023-02-20", "product": "Apple iPhone 12", "quantity": 1, "price": 799.99 }
{ "index": { "_id": "3" } }
{ "order_date": "2023-03-05", "product": "Choco-Lite Bar", "quantity": 25, "price": 0.99 }
{ "index": { "_id": "4" } }
{ "order_date": "2023-04-10", "product": "Nike Air Max 270", "quantity": 3, "price": 150.00 }
{ "index": { "_id": "5" } }
{ "order_date": "2023-05-18", "product": "Samsung Galaxy S21", "quantity": 2, "price": 699.99 }
{ "index": { "_id": "6" } }
{ "order_date": "2023-06-22", "product": "Yoo-hoo Beverage", "quantity": 15, "price": 1.99 }
{ "index": { "_id": "7" } }
{ "order_date": "2023-07-03", "product": "Choco-Lite Bar", "quantity": 30, "price": 0.99 }
{ "index": { "_id": "8" } }
{ "order_date": "2023-08-25", "product": "Apple iPhone 12", "quantity": 1, "price": 799.99 }
{ "index": { "_id": "9" } }
{ "order_date": "2023-09-10", "product": "Nike Air Max 270", "quantity": 4, "price": 150.00 }
{ "index": { "_id": "10" } }
{ "order_date": "2023-10-15", "product": "Samsung Galaxy S21", "quantity": 1, "price": 699.99 }
```

```
{ "index": { "_id": "11" } }
{ "order_date": "2023-11-20", "product": "Yoo-hoo Beverage", "quantity": 20, "price": 1.99 }
{ "index": { "_id": "12" } }
{ "order_date": "2023-12-30", "product": "Choco-Lite Bar", "quantity": 50, "price": 0.99 }
```

2. **Bucket the `order_date` using a Date Histogram Aggregation with Sub-Aggregation:**

   - Use a `date_histogram` to create monthly buckets and a `sum` sub-aggregation to calculate total sales within each month.

```
GET /sales_data/_search
{
  "size": 0,
  "aggs": {
    "sales_over_time": {
      "date_histogram": {
        "field": "order_date",
        "calendar_interval": "month",
        "format": "yyyy-MM"
      },
      "aggs": {
        "total_sales": {
          "sum": {
            "field": "total_value"
          }
        }
      }
    }
  }
}
```

3. **Calculate the Total Value:**

   - Before running the above aggregation, ensure that each document includes a `total_value` field. You could either compute it on the client side or dynamically compute it using an ingest pipeline or a script during the aggregation process.

     For simplicity, let's assume the `total_value` is calculated as `quantity * price`:

```
POST /sales_data/_update_by_query
{
```

```
  "script": {
    "source": "ctx._source.total_value = ctx._source.quantity *
     ↪  ctx._source.price"
  },
  "query": {
    "match_all": {}
  }
}
```

Test

- Run the above `GET /sales_data/_search` query.
- Check the output to see 12 date buckets, one for each month, with the `total_sales` value for each bucket.

Considerations

- The `date_histogram` aggregation is ideal for grouping records by time intervals such as months, weeks, or days.
- The `sum` sub-aggregation allows you to calculate the total value of sales within each date bucket.
- Ensure that the `total_value` field is correctly calculated, as this impacts the accuracy of the sub-aggregation.

Clean-up (Optional)

- Delete the `stocks` index to clean up the data:

```
DELETE /sales_data
```

Documentation

- Date Histogram Aggregation
- Sum Aggregation
- Update By Query API
- Bulk API

2.6 Task: Write and execute a query that searches across multiple clusters

If you are running your instance of Elasticsearch locally, and need to create an additional cluster so that you can run these examples, go to the **Appendix: Adding a Cluster to your Elasticsearch Instance** for information on how to set up an additional single-node cluster.

Example 1: Creating search queries for Products in Multiple Clusters

Requirements

- Set up two single-node clusters on localhost or Elastic Cloud.
- Create an index in each cluster.
- Index at least four documents in each cluster using the _bulk endpoint.
- Configure cross-cluster search.
- Execute a cross-cluster search query.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Set up multiple clusters on localhost.

- Assume you have two clusters, **es01** and **es02** and they have been set up as directed in the **Appendix**.
- In the local cluster, configure communication between the clusters by updating the local cluster settings.

```
PUT /_cluster/settings
{
  "persistent": {
    "cluster": {
      "remote": {
        "es01": {
          "seeds": [
            "es01:9300"
          ],
          "skip_unavailable": true
        },
        "es02": {
```

```
          "seeds": [
            "es02:9300"
          ],
          "skip_unavailable": false
        }
      }
    }
  }
}
```

3. Create a product index in each cluster.

- From the **Kibana Console** (**es01**)

```
PUT /products
{
  "mappings": {
    "properties": {
      "product": {
        "type": "text"
      },
      "category": {
        "type": "keyword"
      },
      "price": {
        "type": "double"
      }
    }
  }
}
```

- From the command line (**es02**).

```
curl -u elastic:[your password here] -X PUT
 ↪  "http://localhost:9201/products?pretty" -H 'Content-Type:
 ↪  application/json' -d'
{
  "mappings": {
    "properties": {
      "product": {
        "type": "text"
      },
      "category": {
```

```
        "type": "keyword"
      },
      "price": {
        "type": "double"
      }
    }
  }
}'
```

4. Index product documents into each cluster.

- For **es01**:

```
POST /products/_bulk
{ "index": { "_id": "1" } }
{ "product": "Elasticsearch Guide", "category": "Books", "price": 29.99 }
{ "index": { "_id": "2" } }
{ "product": "Advanced Elasticsearch", "category": "Books", "price": 39.99 }
{ "index": { "_id": "3" } }
{ "product": "Elasticsearch T-shirt", "category": "Apparel", "price": 19.99 }
{ "index": { "_id": "4" } }
{ "product": "Elasticsearch Mug", "category": "Apparel", "price": 12.99 }
```

- For **es02** through the command line (note that the final single quote is on a line by itself):

```
curl -u elastic:[your password here] -X POST "http://localhost:9201/products/_bulk?pretty" -H 'Content-Type: application/json' -d'
{ "index": { "_id": "5" } }
{ "product": "Elasticsearch Stickers", "category": "Accessories", "price": 4.99 }
{ "index": { "_id": "6" } }
{ "product": "Elasticsearch Notebook", "category": "Stationery", "price": 7.99 }
{ "index": { "_id": "7" } }
{ "product": "Elasticsearch Pen", "category": "Stationery", "price": 3.49 }
{ "index": { "_id": "8" } }
```

```
{ "product": "Elasticsearch Hoodie", "category": "Apparel", "price": 45.99 }
'
```

5. Configure Cross-Cluster Search (CCS).

- In the local cluster, ensure the remote cluster is configured by checking the settings:

```
GET /_cluster/settings?include_defaults=true&filter_path=defaults.cluster.remote
```

6. Execute a Cross-Cluster Search query.

```
GET /products,es02:products/_search
{
  "query": {
    "match": {
      "product": "Elasticsearch"
    }
  }
}
```

Test

1. Verify the index creation.

```
GET /products
```

From the command line execute:

```
curl -u elastic:[your password here] -X GET "http://localhost:9201/products?pretty"
```

2. Verify that the documents have been indexed.

```
GET /products/_search
GET /es02:products/_search
```

3. Ensure the remote cluster is correctly configured and visible from the local cluster.

```
GET /_remote/info
```

4. Execute a Cross-Cluster Search query.

```
GET /products,es02:products/_search
{
  "query": {
    "match": {
      "product": "Elasticsearch"
    }
  }
}
```

Considerations

- Cross-cluster search is useful for querying data across multiple Elasticsearch clusters, providing a unified search experience.
- Ensure the remote cluster settings are correctly configured in the cluster settings.
- Properly handle the index names to avoid conflicts and ensure clear distinction between clusters.

Clean-up (optional)

- Delete the **es01** index.

```
DELETE products
```

- Delete the **es02** index from the command line.

```
curl -u elastic:[your password here] -X DELETE
 ↪  "http://localhost:9201/products?pretty"
```

Documentation

- Bulk API
- Cross-Cluster Search
- Create Index API
- Index Document API

2.7 Task: Write and execute a search that utilizes a runtime field

Example 1: Creating search queries for products with a runtime field for discounted prices

Requirements

- Create an index.
- Index four documents.
- Define a runtime field.
- Execute a search query that creates a query-time runtime field with a 10% discount

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create an index.

```
PUT /product_index
{
  "mappings": {
    "properties": {
      "product": {
        "type": "text"
      },
      "price": {
        "type": "double"
      },
      "category": {
        "type": "keyword"
      }
    }
  }
}
```

3. Index some documents.

```
POST /product_index/_bulk
{ "index": { "_id": "1" } }
{ "product": "Elasticsearch Guide", "price": 29.99, "category": "Books" }
{ "index": { "_id": "2" } }
```

```
{ "product": "Advanced Elasticsearch", "price": 39.99, "category": "Books" }
{ "index": { "_id": "3" } }
{ "product": "Elasticsearch T-shirt", "price": 19.99, "category": "Apparel" }
{ "index": { "_id": "4" } }
{ "product": "Elasticsearch Mug", "price": 12.99, "category": "Apparel" }
```

4. Define a query-time runtime field to return a discounted price.

```
GET product_index/_search
{
  "query": {
    "match_all": {}
  },
  "fields": [
    "product", "price", "discounted_price"
  ],
  "runtime_mappings": {
    "discounted_price": {
      "type": "double",
      "script": {
        "source": "emit(doc['price'].value * 0.9)"
      }
    }
  }
}
```

Test

1. Verify the creation of the index and its mappings.

```
GET /product_index
```

2. Verify the indexed documents.

```
GET /product_index/_search
```

3. Execute the query and confirm the `discounted_price`.

```
{
  ...
    "hits": [
      {
```

```
      ...
      "fields": {
        "product": [
          "Elasticsearch Guide"
        ],
        "price": [
          29.99
        ],
        "discounted_price": [
          26.991
        ]
      }
    },
    {
      ...
      "fields": {
        "product": [
          "Advanced Elasticsearch"
        ],
        "price": [
          39.99
        ],
        "discounted_price": [
          35.991
        ]
      }
    },
    {
      ...
      "fields": {
        "product": [
          "Elasticsearch T-shirt"
        ],
        "price": [
          19.99
        ],
        "discounted_price": [
          17.991
        ]
      }
    },
    {
```

```
        ...
        "fields": {
          "product": [
            "Elasticsearch Mug"
          ],
          "price": [
            12.99
          ],
          "discounted_price": [
            11.691
          ]
        }
      }
    ]
  }
}
```

Considerations

- Runtime fields allow for dynamic calculation of field values at search time, useful for complex calculations or when the field values are not stored.
- The script in the runtime field calculates the discounted price by applying a 10% discount to the price field.

Clean-up (optional)

- Delete the index.

```
DELETE product_index
```

Documentation

- Create Index API
- Bulk API
- Index Document API
- Runtime Fields

Example 2: Creating search queries for employees with a calculated total salary

In this example, the runtime field is defined as part of the index that executes code when documents are indexed. The salary field is read at index time to create a new value for the runtime field total_salary.

Requirements

- An index (`employees`) with documents containing employee information (`name`, `department`, `salary`)and a runtime field (`total_salary`) to calculate the total salary of each employee.
- A search query to retrieve employees with a total salary above $65,000.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the employees index with a mapping for the runtime field.

```
PUT employees
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "department": {
        "type": "text"
      },
      "salary": {
        "type": "integer"
      },
      "total_salary": {
        "type": "long",
        "script": {
          "source": "emit(doc['salary'].value * 12)"
        }
      }
    }
  }
}
```

3. Index some documents that contain a monthly salary.

```
POST /employees/_bulk
{ "index": { "_id": "1" } }
{ "name": "John Doe", "department": "Sales", "salary": 4000 }
{ "index": { "_id": "2" } }
{ "name": "Jane Smith", "department": "Marketing", "salary": 6000 }
{ "index": { "_id": "3" } }
{ "name": "Bob Johnson", "department": "IT", "salary": 7000 }
{ "index": { "_id": "4" } }
{ "name": "Alice Brown", "department": "HR", "salary": 5000 }
```

4. Execute a search query with a runtime field.

```
GET employees/_search
{
  "query": {
    "range": {
      "total_salary": {
        "gte": 65000
      }
    }
  },
  "fields": [
    "total_salary"
  ]
}
```

Test

1. Verify the creation of the index and its mappings.

```
GET /employees
```

2. Verify the indexed documents.

```
GET /employees/_search
```

3. Execute the query and verify the search results contain only employees with a total salary above 65000.

```
{
  ...
    "hits": [
```

```
      {
        "_index": "employees",
        "_id": "2",
        "_score": 1,
        "_source": {
          "name": "Jane Smith",
          "department": "Marketing",
          "salary": 6000
        },
        "fields": {
          "total_salary": [
            72000
          ]
        }
      },
      {
        "_index": "employees",
        "_id": "3",
        "_score": 1,
        "_source": {
          "name": "Bob Johnson",
          "department": "IT",
          "salary": 7000
        },
        "fields": {
          "total_salary": [
            84000
          ]
        }
      }
    ]
  }
}
```

Considerations

- Runtime fields are calculated on the fly and can be used in search queries, aggregations, and sorting.
- The script used in the runtime field calculates the total salary by multiplying the monthly salary by 12 months.

Clean-up (optional)

- Delete the index.

```
DELETE employees
```

Documentation

- Map a Runtime Field
- Script Fields

Example 3: Creating search queries with a runtime field for restaurant data

Requirements

- Create a search query for restaurants in New York City.
- Include the restaurant's `name`, `cuisine`, and a calculated `rating_score` in the search results.
  - the `rating_score` is calculated by taking the square root of the product of the `review_score` and `number_of_reviews`.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create a `restaurant` index.

```
PUT restaurants
{
  "mappings": {
    "properties": {
      "city": {
        "type": "keyword"
      },
      "cuisine": {
        "type": "text"
      },
      "name": {
        "type": "text"
      },
      "number_of_reviews": {
        "type": "long"
      },
      "review_score": {
        "type": "float"
      },
      "state": {
        "type": "keyword"
      }
    }
  }
}
```

3. Index some sample restaurant documents.

```
POST /restaurants/_bulk
{ "index": { "_id": 1 } }
{ "name": "Tasty Bites", "city": "New York", "state": "NY", "cuisine": "Italian", "review_score": 4.5, "number_of_reviews": 200 }
{ "index": { "_id": 2 } }
{ "name": "Spicy Palace", "city": "Los Angeles", "state": "CA", "cuisine": "Indian", "review_score": 4.2, "number_of_reviews": 150 }
{ "index": { "_id": 3 } }
{ "name": "Sushi Spot", "city": "San Francisco", "state": "CA", "cuisine": "Japanese", "review_score": 4.7, "number_of_reviews": 300 }
{ "index": { "_id": 4 } }
{ "name": "Burger Joint", "city": "Chicago", "state": "IL", "cuisine": "American", "review_score": 3.8, "number_of_reviews": 100 }
```

4. Create a query to return restaurants based from New York City.

```
GET restaurants/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "term": {
            "city": {
              "value": "New York"
            }
          }
        },
        {
          "term": {
            "state": {
              "value": "NY"
            }
          }
        }
      ]
    }
  }
}
```

5. Define a runtime field named `weighted_rating` to calculate a weighted rating score for

New York restaurants.

```
GET restaurants/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "term": {
            "city": {
              "value": "New York"
            }
          }
        },
        {
          "term": {
            "state": {
              "value": "NY"
            }
          }
        }
      ]
    }
  },
  "runtime_mappings": {
    "rating_score": {
      "type": "double",
      "script": {
        "source": "emit(Math.sqrt(doc['review_score'].value *
        ↪  doc['number_of_reviews'].value))"
      }
    }
  },
  "fields": [
    "rating_score"
  ]
}
```

Test

- Verify the creation of the index and its mappings.

```
GET /restaurants
```

- Verify the indexed documents.

```
GET /restaurants/_search
```

- Execute the query and verify the restaurant name, cuisine type, and the calculated weighted rating score for restaurants located in New York, NY.

```
{
  ...
    "hits": [
      {
        "_index": "restaurants",
        "_id": "1",
        "_score": 2.4079456,
        "_source": {
          "name": "Tasty Bites",
          "city": "New York",
          "state": "NY",
          "cuisine": "Italian",
          "review_score": 4.5,
          "number_of_reviews": 200
        },
        "fields": {
          "rating_score": [
            30
          ]
        }
      }
    ]
  }
}
```

Considerations

- The runtime_mappings section defines a new field `weighted_rating` that calculates a weighted rating score based on the `review_score` and `number_of_reviews` fields.
- The `query` section uses the `term` query to search for restaurants in New York, NY.
- The `fields` section specifies the fields to include in the search results (in this case, the runtime field `weighted_rating`).

Clean-up (optional)

- Delete the index.

```
DELETE restaurants
```

Documentation

- Runtime Fields in the Search Request

3 Developing Search Applications

3.1 Task: Highlight the search terms in the response of a query

Example 1: Creating search queries w/highlighting for blog posts

Requirements

- Perform a search query which highlights the search term “elasticsearch”

Steps

1. **Open the Kibana Console** or Use a REST Client
2. Create and populate the Index

```
POST /blog_posts/_bulk
{ "index": { "_id": "1" } }
{ "title": "Introduction to Elasticsearch", "content": "Elasticsearch is a powerful search engine." }
{ "index": { "_id": "2" } }
{ "title": "Advanced Elasticsearch Techniques", "content": "This guide covers advanced features of Elasticsearch." }
{ "index": { "_id": "3" } }
{ "title": "Elasticsearch Performance Tuning", "content": "Learn how to optimize Elasticsearch for better performance." }
```

3. Create a search query using the `highlight` clause (the field being searched must match the field to be highlighted)

```
GET /blog_posts/_search
{
  "query": {
    "match": {
      "content": "elasticsearch"
    }
  },
```

```
  "highlight": {
    "fields": {
      "content": {}
    }
  }
}
```

Test

1. Confirm the index exists

```
GET /blog_posts
```

2. Execute the query and confirm that the content field has highlighting

```
{
  ...
  "hits": {
    "hits": [
      {
        "_id": "1",
        "_source": {
          "title": "Introduction to Elasticsearch",
          "content": "Elasticsearch is a powerful search engine."
        },
        "highlight": {
          "content": [
            "<em>Elasticsearch</em> is a powerful search engine."
          ]
        }
      }
      // Additional documents...
    ]
  }
}
```

Considerations

- **Field Selection**: The `highlight` field in the search request specifies which fields to highlight. In this example, we highlight the `content` field.

- **Performance**: Highlighting can impact search performance, especially on large datasets. It is essential to balance the need for highlighting with performance considerations.

Clean-up (optional)

- Delete the index

```
DELETE blog_posts
```

Documentation

- Bulk API
- Highlighting
- Match Query

Example 2: Creating search queries w/highlighting for customer order data

Requirements

- An `orders` index with documents containing customer order information including `customer_name`, `order_date`, `products`, `total_price`.
- A search query to retrieve orders
  - Search for **Product A** and a range for `price` called `total_price`
  - Highlight the search terms in the `products` nested object

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the `orders` index by indexing some documents

```
POST /orders/_bulk
{ "index": { "_id": "1" } }
{ "customer_name": "John Doe", "order_date": "2022-01-01", "products": [{ "name": "Product A", "price": 10.99 },{ "name": "Product B", "price": 5.99 }], "total_price": 16.98 }
{ "index": { "_id": "2" } }
{ "customer_name": "Jane Smith", "order_date": "2022-01-15", "products": [{ "name": "Product B", "price": 5.99 },{ "name": "Product C", "price": 7.99 }], "total_price": 13.98 }
{ "index": { "_id": "3" } }
{ "customer_name": "Bob Johnson", "order_date": "2022-02-01", "products": [{ "name": "Product A", "price": 10.99 },{ "name": "Product C", "price": 7.99 }], "total_price": 18.98 }
```

3. Execute a search query with highlighting including custom `pre_tags` and `post_tags`

```
GET orders/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match_phrase": {
            "products.name": "product a"
          }
        },
        {
```

```
          "range": {
            "total_price": {
              "gt": 10
            }
          }
        }
      ]
    }
  },
  "highlight": {
    "fields": {
      "products.name": {
        "pre_tags": [
          "<b>"
        ],
        "post_tags": [
          "</b>"
        ]
      }
    }
  }
}
```

Test

- Confirm the index exists

```
GET /orders
```

- Execute the query and confirm that `products.name` has highlighting

```
{
  ...
    "hits": [
      {
        "_index": "orders",
        "_id": "1",
        "_score": 1.603535,
        "_source": {
          "customer_name": "John Doe",
          "order_date": "2022-01-01",
          "products": [
```

```
            {
              "name": "Product A", "price": 10.99
            },
            {
              "name": "Product B", "price": 5.99
            }
          ],
          "total_price": 16.98
        },
        "highlight": {
          "products.name": [
            "<b>Product A</b>"
          ]
        }
      },
      {
        "_index": "orders",
        "_id": "3",
        "_score": 1.603535,
        "_source": {
          "customer_name": "Bob Johnson",
          "order_date": "2022-02-01",
          "products": [
            {
              "name": "Product A", "price": 10.99
            },
            {
              "name": "Product C", "price": 7.99
            }
          ],
          "total_price": 18.98
        },
        "highlight": {
          "products.name": [
            "<b>Product A</b>"
          ]
        }
      }
    ]
  }
}
```

Considerations

- Highlighting is used to emphasize the search terms in the response, making it easier to see why a document matched the query.
- The highlight section in the search query specifies which fields to highlight and how to format the highlighted text.
- Nested objects (`products`) are highlighted using the fields section with dot notation (`products.name`, `products.price`).

Clean-up (optional)

- Delete the index

```
DELETE orders
```

Documentation

- Bulk API
- Highlighting
- Match Query
- Nested Objects
- Search API

3.2 Task: Sort the results of a query by a given set of requirements

Example 1: Creating Search Queries w/ Sorting for e-commerce products

Requirements

- Search for e-commerce product data in an index named `products`.
- Sort the results by two criteria:
  - **Primary Sort:** In descending order by product `price` (highest to lowest).
  - **Secondary Sort:** In ascending order by product `name` (alphabetically).

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the `products` index

```
PUT products
{
  "mappings": {
    "properties": {
      "name": {
        "type": "keyword"
      },
      "price": {
        "type": "float"
      }
    }
  }
}
```

3. Index some documents

```
PUT /products/_bulk
{"index":{},"action":"index","_id":"1"}
{"name":"Headphones","price":79.99}
{"index":{},"action":"index","_id":"2"}
{"name":"Smartwatch","price":249.99}
{"index":{},"action":"index","_id":"3"}
{"name":"Laptop","price":1299.99}
{"index":{},"action":"index","_id":"4"}
{"name":"Wireless Speaker","price":99.99}
```

4. Define a query to `match_all` and then perform the primary sort of `price` highest to lowest.

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

5. Define a query to perform the secondary sort of `name` in alphabetical order (`asc`).

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

6. Combine the two sorts and their impact on the results (try the sort with `name` first and `price` second and see how the results change)

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

```
        "order": "desc"
      }
    },
    {
      "name": {
        "order": "asc"
      }
    }
  ]
}
```

Test

- Confirm the index exists

```
GET /products
```

- Run the search queries and examine the response

```
{
  ...
    "hits": [
      {
        "_index": "products",
        "_id": "nXpN-pABRRh1FLFi7Ks8",
        "_score": null,
        "_source": {
          "name": "Laptop",
          "price": 1299.99
        },
        "sort": [
          1299.99,
          "Laptop"
        ]
      },
      {
        "_index": "products",
        "_id": "nHpN-pABRRh1FLFi7Ks8",
        "_score": null,
        "_source": {
          "name": "Smartwatch",
          "price": 249.99
```

```
        },
        "sort": [
          249.99,
          "Smartwatch"
        ]
      },
      {
        "_index": "products",
        "_id": "nnpN-pABRRh1FLFi7Ks8",
        "_score": null,
        "_source": {
          "name": "Wireless Speaker",
          "price": 99.99
        },
        "sort": [
          99.99,
          "Wireless Speaker"
        ]
      },
      {
        "_index": "products",
        "_id": "m3pN-pABRRh1FLFi7Ks8",
        "_score": null,
        "_source": {
          "name": "Headphones",
          "price": 79.99
        },
        "sort": [
          79.99,
          "Headphones"
        ]
      }
    ]
  }
}
```

Considerations

- The `sort` clause defines the sorting criteria.
- An array of sort definitions is specified, prioritizing them from top to bottom.

- In this example, `price` is sorted in descending order (`desc`), while `name` is sorted in ascending order (`asc`).

Clean-up (optional)

- Delete the index

```
DELETE products
```

Documentation

- Sort Search Results

3.3 Task: Implement pagination of the results of a search query

There is only one example here as pagination is rather simple with very few configuration options.

Example 1: Creating pagination queries for an e-commerce product catalog

Requirements

- An index named `products` with documents containing fields like `name`, `price`, `category`, `description`, etc.
- Implement pagination to retrieve search results in batches of 2 documents at a time.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Index sample `products` documents

```
POST /products/_bulk
{"index":{"_id":1}}
{"name":"Product A","price":99.99,"category":"Electronics","description":"High-quality product"}
{"index":{"_id":2}}
{"name":"Product B","price":49.99,"category":"Books","description":"Best-selling novel"}
{"index":{"_id":3}}
{"name":"Product C","price":149.99,"category":"Electronics","description":"Top-rated gadget"}
{"index":{"_id":4}}
{"name":"Product D","price":29.99,"category":"Clothing","description":"Stylish t-shirt"}
{"index":{"_id":5}}
{"name":"Product E","price":19.99,"category":"Books","description":"Classic literature"}
```

3. Define the initial search query with pagination (notice the use of `size`)

```
GET products/_search
{
  "size": 2,
  "query": {
    "match_all": {}
  },
  "from": 0
}
```

4. To retrieve the next page of results, use one of two methods:

    1. Update the `from` field with the document count to proceed from (not the document id)

```
GET products/_search
{
  "size": 2,
  "query": {
    "match_all": {}
  },
  "from": 2
}
```

    2. **OR** If you are sorting the documents as well as paginating then you can use the `search_after` parameter along with the sort values from the last hit in the previous page

```
// search with sort on page 1
GET /products/_search
{
  "query": {
    "match_all": {}
  },
  "sort": [
    "_doc"
  ],
  "size": 2
}
```

```
// second search using the sort value
// from the last document of the previous search
GET /products/_search
```

```
{
  "query": {
    "match_all": {}
  },
  "sort": [
    "_doc"
  ],
  "size": 2,
  "search_after": [6]
}
```

Test

1. Confirm the index exists

```
GET /products
```

2. Execute the initial search query to retrieve the first 2 documents

```
{
  ...
    "hits": [
      {
        "_index": "products",
        "_id": "1",
        "_score": 1,
        "_source": {
          "name": "Product A",
          "price": 99.99,
          "category": "Electronics",
          "description": "High-quality product"
        }
      },
      {
        "_index": "products",
        "_id": "2",
        "_score": 1,
        "_source": {
          "name": "Product B",
          "price": 49.99,
          "category": "Books",
          "description": "Best-selling novel"
```

```
        }
      }
    ]
  }
}
```

3. Change `from` to **2** and execute it again to get the next 2 items.

```
{
  ...
    "hits": [
      {
        "_index": "products",
        "_id": "3",
        "_score": 1,
        "_source": {
          "name": "Product C",
          "price": 149.99,
          "category": "Electronics",
          "description": "Top-rated gadget"
        }
      },
      {
        "_index": "products",
        "_id": "4",
        "_score": 1,
        "_source": {
          "name": "Product D",
          "price": 29.99,
          "category": "Clothing",
          "description": "Stylish t-shirt"
        }
      }
    ]
  }
}
```

Considerations

- The `size` parameter specifies the number of documents to retrieve per page.
- The `from` parameter is used for the initial query to start from the beginning.

- The `search_after` parameter can be used for subsequent queries to retrieve the next page of results based on the sort values from the last hit or simply update the `from` parameter to start with the next group starting from a certain number of items in the search results. The following are required to use `search_after`
  - The `sort` parameter is used to ensure consistent ordering of results across pages.
  - The `_doc` field is used as a tiebreaker to ensure a stable sort order.

Clean-up (optional)

- Delete the index

```
DELETE products
```

Documentation

- Pagination
- Search After API
- Sort Search Results

3.4 Task: Define and use index aliases

Example 1: Creating Index Aliases for Customer Data

This is an example of the simplest kind of alias.

Requirements

- Create multiple indices for customer data (e.g., `customers-2024-01`, `customers-2024-02`).
- Create an alias that points to these indices.
- Use the alias to perform search operations across all customer indices.

Steps

1. **Open the Kibana Console** or use a REST Client.
2. Create the 2 indices by indexing sample documents

```
POST /customers-2024-01/_bulk
{ "index": { "_id": "1" } }
{ "name": "John Doe", "email": "john.doe@example.com", "signup_date":
 ↪  "2024-01-15" }
{ "index": { "_id": "2" } }
{ "name": "Jane Smith", "email": "jane.smith@example.com",
 ↪  "signup_date": "2024-01-20" }
```

```
POST /customers-2024-02/_bulk
{ "index": { "_id": "1" } }
{ "name": "Alice Johnson", "email": "alice.johnson@example.com",
 ↪  "signup_date": "2024-02-05" }
{ "index": { "_id": "2" } }
{ "name": "Bob Brown", "email": "bob.brown@example.com", "signup_date":
 ↪  "2024-02-10" }
```

3. Create an alias for the two indices

```
POST _aliases
{
  "actions": [
    {
      "add": {
        "index": "customers-*",
```

```
        "alias": "customers"
      }
    }
  ]
}
```

4. Execute a search query using the alias and confirm 4 documents returned

```
GET /customers/_search
```

5. Execute a search query for John Doe's record

```
GET customers/_search
{
  "query": {
    "match": {
      "name": "john doe"
    }
  }
}
```

Test

1. Verify the alias was created

```
GET /_alias/customers
```

2. Confirm 4 documents returned when executing the test query

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

Or just

```
GET /customers/_search
```

Considerations

- **Alias Flexibility**: Using an alias allows for flexibility in managing indices. The alias can point to multiple indices, making it easier to manage and query data across time-based indices.
- **Index Patterns**: Ensure that the alias name (`customers`) is descriptive and clearly indicates its purpose.
- **Performance**: Searching using an alias is efficient and does not introduce significant overhead compared to searching directly on indices.

Clean-up (optional)

- Delete the aliases

```
DELETE customers-2024-01/_alias/customers
DELETE customers-2024-02/_alias/customers
```

- Delete the 2 indices

```
DELETE customers-2024-01
DELETE customers-2024-02
```

Documentation

- Bulk API
- Delete Aliases
- Index Aliases
- Search API

Example 2: Creating index aliases for logging data with filtering and routing

This is a slightly more complex use of an index alias. It includes a custom configuration for each index defined in the alias and any custom **filtering** and/or **routing** that is required.

Requirements

- Three indices (`logs_2022`, `logs_2023`, and `logs_2024`) with documents containing log data (`message`, `level`, `timestamp`)
- An index alias (`logs`) that points to all three indices with filtering and routing based on the log level
  - `logs_2022`
    - `filter` on `level` **ERROR**
    - `routing` to **error**
  - `logs_2023`
    - `filter` on `level` **INFO**
    - `routing` to **info**
  - `logs_2024`
    - `filter` on `level` **DEBUG**
    - `routing` to **debug**
- A search query against the message field to retrieve documents from the alias

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the `logs_2022`, `logs_2023`, and `logs_2024` indices

```
PUT logs_2022
{
  "mappings": {
    "properties": {
      "message": {
        "type": "text"
      },
      "level": {
        "type": "keyword"
      },
      "timestamp": {
        "type": "date"
```

```
      }
    }
  }
}
```

```
PUT logs_2023
{
  "mappings": {
    "properties": {
      "message": {
        "type": "text"
      },
      "level": {
        "type": "keyword"
      },
      "timestamp": {
        "type": "date"
      }
    }
  }
}
```

```
PUT logs_2024
{
  "mappings": {
    "properties": {
      "message": {
        "type": "text"
      },
      "level": {
        "type": "keyword"
      },
      "timestamp": {
        "type": "date"
      }
    }
  }
}
```

3. Create an index alias (`logs`) with filtering and routing (this must be done before indexing any documents)

```
POST /_aliases
{
  "actions": [
    {
      "add": {
        "index": "logs_2022",
        "alias": "logs",
        "filter": {
          "term": {
            "level": "ERROR"
          }
        },
        "routing": "error"
      }
    },
    {
      "add": {
        "index": "logs_2023",
        "alias": "logs",
        "filter": {
          "term": {
            "level": "INFO"
          }
        },
        "routing": "info"
      }
    },
    {
      "add": {
        "index": "logs_2024",
        "alias": "logs",
        "filter": {
          "term": {
            "level": "DEBUG"
          }
        },
        "routing": "debug"
      }
    }
  ]
}
```

4. Index sample documents

```
POST /logs_2022/_bulk
{ "index": { "_id": "1" } }
{ "message": "Error occurred", "level": "ERROR", "timestamp":
 ↪  "2022-01-01T12:00:00Z" }
{ "index": { "_id": "2" } }
{ "message": "Error occurred", "level": "ERROR", "timestamp":
 ↪  "2022-01-01T12:00:00Z" }
```

```
POST /logs_2023/_bulk
{ "index": { "_id": "1" } }
{ "message": "Info message", "level": "INFO", "timestamp":
 ↪  "2023-01-01T12:00:01Z" }
{ "index": { "_id": "2" } }
{ "message": "Info message", "level": "INFO", "timestamp":
 ↪  "2023-01-01T12:00:01Z" }
```

```
POST /logs_2024/_bulk
{ "index": { "_id": "1" } }
{ "message": "Debug message", "level": "DEBUG", "timestamp":
 ↪  "2024-01-01T12:00:01Z" }
{ "index": { "_id": "2" } }
{ "message": "Debug message", "level": "DEBUG", "timestamp":
 ↪  "2024-01-01T12:00:01Z" }
```

5. Create a general search using the `logs` alias (all log messages should be returned)

```
GET logs/_search
```

6. Create a search query using the `logs` alias to search for **error** and **info** messages

```
GET /logs/_search
{
  "query": {
    "terms": {
      "message": [
        "error",
        "info"
      ]
    }
  }
}
```

Test

- Verify the alias was created

```
GET /_alias/logs
```

- Confirm 4 documents returned when executing the query from Step 6: 2 from `logs_2022` and 2 from `logs_2023`

```
{
  ...
    "hits": [
      {
        "_index": "logs_2022",
        "_id": "1",
        "_score": 1,
        "_source": {
          "message": "Error occurred",
          "level": "ERROR",
          "timestamp": "2022-01-01T12:00:00Z"
        }
      },
      {
        "_index": "logs_2022",
        "_id": "2",
        "_score": 1,
        "_source": {
          "message": "Error occurred",
          "level": "ERROR",
          "timestamp": "2022-01-01T12:00:00Z"
        }
      },
      {
        "_index": "logs_2023",
        "_id": "1",
        "_score": 1,
        "_source": {
          "message": "Info message",
          "level": "INFO",
          "timestamp": "2023-01-01T12:00:01Z"
        }
      },
      {
```

```
        "_index": "logs_2023",
        "_id": "2",
        "_score": 1,
        "_source": {
          "message": "Info message",
          "level": "INFO",
          "timestamp": "2023-01-01T12:00:01Z"
        }
      }
    ]
  }
}
```

Considerations

- The index must be set up in the proper order for the query using the alias with `filter` and `routing` to work:
  - create the index
  - create the alias using filtering and/or routing
  - index the documents
- Index aliases with filtering and routing allow you to control which documents are included in the alias based on specific criteria.
- In this example, we created an alias that points to three indices with filtering based on the log level and routing to separate indices.

Clean-up (optional)

- Delete the aliases

```
DELETE logs_2022/_alias/logs
DELETE logs_2023/_alias/logs
DELETE logs_2024/_alias/logs
```

- Delete the indices

```
DELETE logs_2022
DELETE logs_2023
DELETE logs_2024
```

Documentation

- Bulk API
- Delete Aliases
- Index Aliases
- Search API

3.5 Task: Define and use a search template

Example 1: Creating Search Templates for Product Search

Requirements

- Create an index and populate it with example product documents
- Define a search template for querying products using `description`

Steps

1. **Open the Kibana Console** or use a REST Client
2. Create the index as a side-effect of indexing sample documents

```
POST /products/_bulk
{ "index": { "_id": "1" } }
{ "name": "Laptop", "description": "A high-performance laptop", "price": 999.99 }
{ "index": { "_id": "2" } }
{ "name": "Smartphone", "description": "A latest model smartphone", "price": 799.99 }
{ "index": { "_id": "3" } }
{ "name": "Tablet", "description": "A new tablet with excellent features", "price": 499.99 }
```

3. Define a search template

```
PUT _scripts/product_search_template
{
  "script": {
    "lang": "mustache",
    "source": {
      "query": {
        "match": {
          "description": "{{query_string}}"
        }
      }
    }
  }
}
```

4. Use the search template

```
GET products/_search/template
{
  "id": "product_search_template",
  "params": {
    "query_string": "laptop"
  }
}
```

Test

1. Verify the documents are indexed

```
GET products/_search
```

2. Verify the template was created

```
GET _scripts/product_search_template
```

3. Perform a search using the template (results below)

```
{
  "hits": {
    "total": {
      "value": 1,
      "relation": "eq"
    },
    "hits": [
      {
        "_index": "products",
        "_id": "1",
        "_source": {
          "name": "Laptop",
          "description": "A high-performance laptop",
          "price": 999.99
        }
      }
    ]
  }
}
```

Considerations

- **Template Flexibility**: Using a search template allows for reusable and parameterized queries, reducing the need to write complex queries multiple times.
- **Performance**: Search templates can improve performance by reusing the query logic and reducing the overhead of constructing queries dynamically.
- **Template Language**: Mustache is used as the templating language, providing a simple and powerful way to define dynamic queries.

Clean-up (optional)

- Delete the search template

```
DELETE _scripts/product_search_template
```

- Delete the index

```
DELETE products
```

Documentation

- Bulk API
- Mustache
- Search API
- Search Templates

Example 2: Creating search templates for an e-commerce product catalog with sorting and pagination

Requirements

- An Elasticsearch index named `products` with documents containing the fields `name`, `price`, `category`, `description`, `rating`.
- Define a search template to search for products based on a user-provided query string, category filter, sort order, and pagination.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Index sample product documents using the /_bulk endpoint:

```
POST /products/_bulk
{"index":{"_id":1}}
{"name":"Product A", "price":99.99, "category":"Electronics",
 ↪  "description":"High-quality product", "rating":4.2}
{"index":{"_id":2}}
{"name":"Product B", "price":49.99, "category":"Books",
 ↪  "description":"Best-selling novel", "rating":4.5}
{"index":{"_id":3}}
{"name":"Product C", "price":149.99, "category":"Electronics",
 ↪  "description":"Top-rated gadget", "rating":3.8}
{"index":{"_id":4}}
{"name":"Product D", "price":29.99, "category":"Clothing",
 ↪  "description":"Stylish t-shirt", "rating":4.1}
```

3. Iteratively create a search query that satisfies the various requirements (query string, category filter, sort order, pagination)

```
// just search for a product using the name field
GET products/_search
{
  "query": {
    "query_string": {
      "default_field": "name",
      "query": "product"
    }
  }
}
```

```
// add the filter which entails also changing the query set-up
GET products/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "query_string": {
            "default_field": "name",
            "query": "product"
          }
        }
      ],
      "filter": [
        {
          "term": {
            "category": "electronics"
          }
        }
      ]
    }
  }
}
```

```
// add sort
GET products/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "query_string": {
            "default_field": "name",
            "query": "product"
          }
        }
      ],
      "filter": [
        {
          "term": {
            "category": "electronics"
          }
```

```
        }
      ]
    }
  },
  "sort": [
    {
      "price": {
        "order": "desc"
      }
    }
  ]
}
```

```
// add pagination (such as it is)
GET products/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "query_string": {
            "default_field": "name",
            "query": "product"
          }
        }
      ],
      "filter": [
        {
          "term": {
            "category": "electronics"
          }
        }
      ]
    }
  },
  "sort": [
    {
      "price": {
        "order": "desc"
      }
    }
  ],
```

```
  "size": 1,
  "from": 1
}
```

4. Using the above, define the search template:

```
PUT _scripts/product_search_template
{
  "script": {
    "lang": "mustache",
    "source": {
      "query": {
        "bool": {
          "must": [
            {
              "query_string": {
                "default_field": "name",
                "query": "{{query_string}}"
              }
            }
          ],
          "filter": [
            {
              "term": {
                "category": "{{category}}"
              }
            }
          ]
        }
      },
      "sort": [
        {
          "price": {
            "order": "{{sort_order}}"
          }
        }
      ],
      "from": "{{from}}",
      "size": "{{size}}"
    }
  }
}
```

5. Use the `_render` endpoint to take a look at the formatting of the query

```
POST _render/template
{
  "id": "product_search_template",
  "params": {
    "query_string": "product",
    "category" : "electronics",
    "sort_order" : "desc",
    "from": 0,
    "size": 1
  }
}
```

6. Use the search template with sorting and pagination:

```
GET products/_search/template
{
  "id": "product_search_template",
  "params": {
    "query_string": "product",
    "category" : "books",
    "sort_order" : "desc",
    "from": 0,
    "size": 1
  }
}
```

Test

- Verify the documents are indexed

```
GET products/_search
```

- Verify the template is created

```
GET _scripts/product_search_template
```

- Execute a query for **product**, category of **books**, sort order of **desc**, and pagination starting at item **0**, with a result size of **1**.

```
{
  ...
    "hits": [
```

```
      {
        "_index": "products",
        "_id": "2",
        "_score": null,
        "_source": {
          "name": "Product B",
          "price": 49.99,
          "category": "Books",
          "description": "Best-selling novel",
          "rating": 4.5
        },
        "sort": [
          49.99
        ]
      }
    ]
  }
}
```

Considerations

- The search template includes sorting and pagination parameters (`sort_field`, `sort_order`, `from`, `size`).
- The `sort` parameter in the template specifies the field and order for sorting the results.
- The `from` and `size` parameters control the pagination of the results.
- The `params` object in the search template request provides the values for all placeholders in the template.

Clean-up (optional)

- Delete the search template

```
DELETE _scripts/product_search_template
```

- Delete the index

```
DELETE products
```

Documentation

- Search Template
- Mustache Language
- Paginate Search Results
- Scripting in Elasticsearch
- Sort Search Results

Example 3: Creating search templates for an e-commerce product catalog with nested queries, sorting, pagination, and aggregations

Requirements

- An index named `products` with documents containing the fields `name`, `price`, `category`, `description`, `rating`, `tags`, `specifications`, `specifications.ram` and `specifications.storage`.
- Define a search template to search for products based on:
  - a user-provided query string,
  - category filter
  - tag filter
  - sort order
  - pagination
- include aggregations for `category`, `tags`, and `price_range`

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the index with two fields (`category` and `tags`) of type `keyword` for use in the aggregation

```
PUT products
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "price" : {
        "type": "float"
      },
      "category" : {
        "type": "keyword"
      },
      "description" : {
        "type": "text"
      },
      "rating" : {
        "type": "float"
      },
```

```
      "tags" : {
        "type": "keyword"
      },
      "specifications" : {
        "properties": {
          "ram" : {
            "type" : "text"
          },
          "storage" : {
            "type" : "text"
          }
        }
      }
    }
  }
}
```

3. Index sample `products` documents

```
POST /products/_bulk
{"index":{"_id":1}}
{"name":"Product A", "price":99.99, "category":"Electronics",
 ↪  "description":"High-quality product", "rating":4.2,
 ↪  "tags":["electronics", "gadget"], "specifications":{"ram":"8GB",
 ↪  "storage":"256GB"}}
{"index":{"_id":2}}
{"name":"Product B", "price":49.99, "category":"Books",
 ↪  "description":"Best-selling novel", "rating":4.5, "tags":["book",
 ↪  "fiction"]}
{"index":{"_id":3}}
{"name":"Product C", "price":149.99, "category":"Electronics",
 ↪  "description":"Top-rated gadget", "rating":3.8,
 ↪  "tags":["electronics", "laptop"], "specifications":{"ram":"16GB",
 ↪  "storage":"512GB"}}
{"index":{"_id":4}}
{"name":"Product D", "price":29.99, "category":"Clothing",
 ↪  "description":"Stylish t-shirt", "rating":4.1, "tags":["clothing",
 ↪  "tshirt"]}
```

4. Define a search query that satisfies the requirements

```
GET products/_search
{
```

```
  "query": {
    "bool": {
      "must": [
        {
          "query_string": {
            "default_field": "name",
            "query": "product"
          }
        }
      ],
      "filter": [
        {
          "term": {
            "category": "Electronics"
          }
        },
        {
          "term": {
            "tags": "electronics"
          }
        }
      ]
    }
  },
  "sort": [
    {
      "price": {
        "order": "desc"
      }
    }
  ],
  "size": 1,
  "from": 0,
  "aggs": {
    "category_aggs": {
      "terms": {
        "field": "category"
      }
    },
    "tags_aggs": {
      "terms": {
        "field": "tags"
```

```
      }
    },
    "price_range_aggs": {
      "range": {
        "field": "price",
        "ranges": [
          {
            "to": 30
          },
          {
            "from": 30,
            "to": 100
          },
          {
            "from": 100
          }
        ]
      }
    }
  }
}
```

5. Use the above query to create the search template

```
PUT _scripts/products_search_template
{
  "script": {
    "lang": "mustache",
    "source": {
      "query": {
        "bool": {
          "must": [
            {
              "query_string": {
                "default_field": "name",
                "query": "{{query}}"
              }
            }
          ],
          "filter": [
            {
              "term": {
```

```
                "category": "{{category}}"
              }
            },
            {
              "term": {
                "tags": "{{tags}}"
              }
            }
          ]
        }
      },
      "sort": [
        {
          "price": {
            "order": "{{sort_order}}"
          }
        }
      ],
      "size": "{{size}}",
      "from": "{{from}}",
      "aggs": {
        "category_aggs": {
          "terms": {
            "field": "category"
          }
        },
        "tags_aggs": {
          "terms": {
            "field": "tags"
          }
        },
        "price_range_aggs": {
          "range": {
            "field": "price",
            "ranges": [
              {
                "to": 30
              },
              {
                "from": 30,
                "to": 100
              },
```

```
                {
                  "from": 100
                }
              ]
            }
          }
        }
      }
    }
  }
}
```

6. Use the search template with sorting, pagination, and aggregations:

```
GET products/_search/template
{
  "id": "products_search_template",
  "params": {
    "query" : "product",
    "category" : "Electronics",
    "tags" : "electronics",
    "sort_order" : "desc",
    "size" : 2,
    "from" : 0
  }
}
```

Test

- Verify the index is created

```
GET products
```

- Verify the documents are indexed

```
GET products/_search
```

- Verify the template is created

```
GET _scripts/products_search_template
```

- Execute a search using the search template query, and it should return the first 2 documents matching the provided query string (“*“), category filter (”Electronics”), tag filter (“electronics”), sorted by price in descending order with aggregations.

– The response should also include aggregations for `category`, `tags`, and `price`.

```
{
  ...
  "hits": [
    {
      "_index": "products",
      "_id": "3",
      "_score": null,
      "_source": {
        "name": "Product C",
        "price": 149.99,
        "category": "Electronics",
        "description": "Top-rated gadget",
        "rating": 3.8,
        "tags": [
          "electronics",
          "laptop"
        ],
        "specifications": {
          "ram": "16GB",
          "storage": "512GB"
        }
      },
      "sort": [
        149.99
      ]
    },
    {
      "_index": "products",
      "_id": "1",
      "_score": null,
      "_source": {
        "name": "Product A",
        "price": 99.99,
        "category": "Electronics",
        "description": "High-quality product",
        "rating": 4.2,
        "tags": [
          "electronics",
          "gadget"
        ],
        "specifications": {
```

```
          "ram": "8GB",
          "storage": "256GB"
        }
      },
      "sort": [
        99.99
      ]
    }
  ]
},
"aggregations": {
  "category_aggs": {
    "doc_count_error_upper_bound": 0,
    "sum_other_doc_count": 0,
    "buckets": [
      {
        "key": "Electronics",
        "doc_count": 2
      }
    ]
  },
  "tags_aggs": {
    "doc_count_error_upper_bound": 0,
    "sum_other_doc_count": 0,
    "buckets": [
      {
        "key": "electronics",
        "doc_count": 2
      },
      {
        "key": "gadget",
        "doc_count": 1
      },
      {
        "key": "laptop",
        "doc_count": 1
      }
    ]
  },
  "price_range_aggs": {
    "buckets": [
      {
```

```
        "key": "*-30.0",
        "to": 30,
        "doc_count": 0
      },
      {
        "key": "30.0-100.0",
        "from": 30,
        "to": 100,
        "doc_count": 1
      },
      {
        "key": "100.0-*",
        "from": 100,
        "doc_count": 1
      }
    ]
  }
}
}
```

Considerations

- The search template includes queries, filters, sorting, pagination, and aggregations.
- The `tags` filter uses a `terms` query to match documents with any of the specified tags.
- The `aggs` section in the template defines the aggregations to be included in the search results.
- The `params` object in the search template request provides the values for all placeholders in the template.

Clean-up (optional)

- Delete the search template

```
DELETE _scripts/product_search_template
```

- Delete the index

```
DELETE products
```

Documentation

- Aggregations
- Mustache Language
- Nested Aggregations
- Paginate Search Results
- Scripting in Elasticsearch
- Search Template
- Sort Search Results

4 Data Processing

4.1 Task: Define a mapping that satisfies a given set of requirements

Example 1: Defining Index Mappings for a Product Catalog

Requirements

- Create a mapping for an index named `product_catalog`
- Define fields for product ID, name, description, price, and availability status.
- Ensure the `price` field is a numeric type.
- Use a text type for `description` with a keyword sub-field for exact matches.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Create the index with mappings:

```
PUT /product_catalog
{
  "mappings": {
    "properties": {
      "product_id": {
        "type": "keyword"
      },
      "name": {
        "type": "text"
      },
      "description": {
        "type": "text",
        "fields": {
          "keyword": {
            "type": "keyword",
            "ignore_above": 256
```

```
          }
        }
      },
      "price": {
        "type": "double"
      },
      "availability_status": {
        "type": "boolean"
      }
    }
  }
}
```

3. Create sample documents using the `_bulk` endpoint:

```
POST /product_catalog/_bulk
{ "index": { "_id": "1" } }
{ "product_id": "p001", "name": "Product 1", "description": "Description of product 1", "price": 19.99, "availability_status": true }
{ "index": { "_id": "2" } }
{ "product_id": "p002", "name": "Product 2", "description": "Description of product 2", "price": 29.99, "availability_status": false }
```

Test

1. Retrieve the mappings to verify:

```
GET /product_catalog/_mapping
```

2. Search for documents to confirm they are indexed correctly:

```
GET /product_catalog/_search
```

OR

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

OR

```
GET product_catalog/_search
{
  "query": {
    "term": {
      "description": "product"
    }
  }
}
```

OR

```
GET product_catalog/_search
{
  "query": {
    "match": {
      "description.keyword": "Description of product 1"
    }
  }
}
```

Considerations

- The `price` field is set to `integer` to handle whole numbers.
- The `description` field includes a `keyword` sub-field for exact match searches.

Clean-up (optional)

- Delete the index (which will also delete the mapping)

```
DELETE product_catalog
```

Documentation

- Bulk API
- Elasticsearch Index Mappings

Example 2: Creating a mapping for a social media platform

Requirements

- Create a mapping for an index named `users`
- The mapping should have a field called `username` of type `keyword`
- The mapping should have a field called `email` of type `keyword`
- The mapping should have a field called `posts` of type `array` containing `object` values
- The `posts` array should have a property called `content` of type `text`
- The `posts` array should have a property called `likes` of type `integer`

Steps

1. **Open the Kibana Console** or use a REST client
2. Create an index with the desired mapping:

```
PUT /users
{
  "mappings": {
    "properties": {
      "username": {
        "type": "keyword"
      },
      "email": {
        "type": "keyword"
      },
      "posts": {
        "properties": {
          "content": {
            "type": "text"
          },
          "likes": {
            "type": "integer"
          }
        }
      }
    }
  }
}
```

3. Index a document:

```
POST /users/_doc
{
  "username": "john_doe",
  "email": "john.doe@example.com",
  "posts": [
    {
      "content": "Hello World!",
      "likes": 10
    },
    {
      "content": "This is my second post",
      "likes": 5
    }
  ]
}
```

Test

- Verify the mapping

```
GET users
```

- Use the _search API to verify that the mapping is correct and the data is indexed:

```
GET /users/_search
{
  "query": {
    "match": {
      "username": "john_doe"
    }
  }
}
```

And

```
GET users/_search
{
  "size": 0,
  "aggs": {
    "total_likes": {
      "sum": {
```

```
        "field": "posts.likes"
      }
    }
  }
}
```

Considerations

- The `username` and `email` fields are of type `keyword` to enable exact matching.
- The `posts` field is of type `array` with `object` values to enable storing multiple posts per user.
- The `content` field is of type `text` to enable full-text search.
- The `likes` field is of type `integer` to enable aggregations and sorting.

Clean-up (optional)

- Delete the index (which will also delete the mapping)

```
DELETE users
```

Documentation

- Elasticsearch Index Mappings

Example 3: Creating a mapping for storing and searching restaurant data

Requirements

- Create a mapping for an index named `restaurants`.
- The mapping should include fields for:
  - `name` (`text` field for restaurant name)
  - `description` (`text` field for restaurant description)
  - `location` (geolocation field for restaurant location)

Steps

1. **Open the Kibana Console** or use a REST client
2. Define the mapping using a REST API call:

```
PUT /restaurants
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "description": {
        "type": "text"
      },
      "location": {
        "type": "geo_point"
      }
    }
  }
}
```

Test

1. Verify that the mapping is created successfully by using the following API call:

```
GET /restaurants/_mapping
```

2. Try indexing a sample document with the defined fields:

```
PUT /restaurants/_doc/1
{
  "name": "Pizza Palace",
  "description": "Delicious pizzas and Italian cuisine",
  "location": {
    "lat": 40.7128,
    "lon": -74.0059
  }
}
```

3. Use search queries to test text search on `name` and `description` fields, and utilize geo-queries to search based on the `location` field.

```
GET /restaurants/_search
{
  "query": {
    "match": {
      "name": "Pizza Palace"
    }
  }
}
```

```
GET /restaurants/_search
{
  "query": {
    "match": {
      "description": "Italian cuisine"
    }
  }
}
```

```
GET /restaurants/_search
{
  "query": {
    "bool": {
      "filter": {
        "geo_distance": {
          "distance": "5km",
          "location": {
            "lat": 40.7128,
            "lon": -74.0059
          }
```

```
        }
      }
    }
  }
}
```

Considerations

- `text` is a generic field type suitable for textual data like names and descriptions.
- `geo_point` is a specialized field type for storing and searching geospatial data like **latitude** and **longitude** coordinates.

Clean-up (optional)

- Delete the index (which will also delete the mapping)

```
DELETE restaurants
```

Documentation

- Data Types
- Geolocation Queries

4.2 Task: Define and use a custom analyzer that satisfies a given set of requirements

Example 1: Custom Analyzer for Restaurant Reviews

4.2.0.1 Requirements

- Create a mapping for an index named `restaurant_reviews`
- Create a custom analyzer named `custom_review_analyzer`.
- The analyzer should:
  - Use the `standard` tokenizer.
  - Include a `lowercase` filter.
  - Include a `stop` filter to remove common English stop words.
  - Include a `synonym` filter to handle common synonyms.

4.2.0.2 Steps

1. **Open the Kibana Console** or use a REST client
2. Create the index with a custom analyzer defined in the index settings.

```
PUT /restaurant_reviews
{
  "settings": {
    "analysis": {
      "analyzer": {
        "custom_review_analyzer": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "stop",
            "synonym"
          ]
        }
      },
      "filter": {
        "synonym": {
          "type": "synonym",
          "synonyms": [
            "delicious, tasty",
```

```
            "restaurant, eatery"
          ]
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "review_id": {
        "type": "keyword"
      },
      "restaurant_name": {
        "type": "text"
      },
      "review_text": {
        "type": "text",
        "analyzer": "custom_review_analyzer"
      },
      "rating": {
        "type": "integer"
      },
      "review_date": {
        "type": "date"
      }
    }
  }
}
```

3. Add some sample documents to the index to test the custom analyzer

```
POST /restaurant_reviews/_bulk
{ "index": {} }
{ "review_id": "1", "restaurant_name": "Pizza Palace", "review_text":
 ↪  "The pizza was delicious and the service was excellent.", "rating":
 ↪  5, "review_date": "2024-07-01" }
{ "index": {} }
{ "review_id": "2", "restaurant_name": "Burger Haven", "review_text":
 ↪  "Tasty burgers and friendly staff.", "rating": 4, "review_date":
 ↪  "2024-07-02" }
```

4. Perform a search query to verify the custom analyzer is working as expected.

```
GET /restaurant_reviews/_search
{
  "query": {
    "match": {
      "review_text": "tasty"
    }
  }
}
```

4.2.0.3 Considerations

- **Standard Tokenizer**: Chosen for its ability to handle most text inputs effectively.
- **Lowercase Filter**: Ensures case-insensitive search.
- **Stop Filter**: Removes common stop words to improve search relevance.
- **Synonym Filter**: Handles common synonyms to enhance search matching.

4.2.0.4 Test

1. Verify the analyzer was created

```
GET /restaurant_reviews/_settings
```

2. Verify the custom analyzer configuration using the `_analyze` API to test the custom analyzer directly.

```
GET /restaurant_reviews/_analyze
{
  "analyzer": "custom_review_analyzer",
  "text": "The pizza was delicious and the service was excellent."
}
```

3. Perform a search queries to ensure the custom analyzer processes the text as expected.

```
GET /restaurant_reviews/_search
{
  "query": {
    "match": {
      "review_text": "tasty"
    }
  }
}
```

4.2.0.5 Clean-up (optional)

- Delete the Index

```
DELETE /restaurant_reviews
```

4.2.0.6 Documentation

- Analyzers
- Custom Analyzers
- Index Settings

Example 2: Creating a custom analyzer for product descriptions

Requirements

- Create a mapping for an index named `products` with a `description` field containing product descriptions
- The custom analyzer should:
  - Lowercase all text
  - Remove stop words (common words like `the`, `and`, `a`, etc.)
  - Split text into individual words (tokenize)
  - Stem words (reduce words to their root form, e.g., `running` - `run`)

Steps

1. **Open the Kibana Console** or use a REST client
2. Create the `products` index with a custom analyzer for the `description` field:

```
PUT /products
{
  "settings": {
    "analysis": {
      "analyzer": {
        "product_description_analyzer": {
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "stop",
            "stemmer"
          ]
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "description": {
        "type": "text",
        "analyzer": "product_description_analyzer"
      }
    }
```

```
  }
}
```

3. Index some sample documents using the `_bulk` endpoint:

```
POST /products/_bulk
{ "index": { "_id": 1 } }
{ "description": "The quick brown fox jumps over the lazy dog." }
{ "index": { "_id": 2 } }
{ "description": "A high-quality product for running enthusiasts." }
```

Test

1. Search for documents containing the term **run**

```
GET /products/_search
{
  "query": {
    "match": {
      "description": "run"
    }
  }
}
```

This should return the document with `_id` **2**, as the custom analyzer has stemmed **running** to **run**.

2. Search for documents containing the term **the**

```
GET /products/_search
{
  "query": {
    "match": {
      "description": "the"
    }
  }
}
```

This should not return any documents, as the custom analyzer has removed stop words like **the**.

Considerations

- The custom analyzer is defined in the index settings using the `analysis` section.
- The `tokenizer` parameter specifies how the text should be split into tokens (individual words).
- The `filter` parameter specifies the filters to be applied to the tokens, such as lowercasing, stop word removal, and stemming.
- The custom analyzer is applied to `description` by specifying it in the field mapping.

Clean-up (optional)

- Delete the Index

```
DELETE /products
```

Documentation

- Analyzers
- Custom Analyzers
- Token Filters
- Tokenizers

Example 3: Creating a custom analyzer for product descriptions in an ecommerce catalog

Requirements

- Define an index called `product_catalog` with a `description` field.
- Create a custom tokenizer that splits text on non-letter characters.
- Include a lowercase filter to normalize text.
- Add a stopword filter to remove common English stopwords.

Steps

1. **Open the Kibana Console** or use a REST client
2. Define the custom analyzer in the index settings

```
PUT product_catalog
{
  "settings": {
    "analysis": {
      "analyzer": {
        "custom_analyzer": {
          "type": "custom",
          "tokenizer": "lowercase",
          "filter": [
            "english_stop"
          ]
        }
      },
      "filter": {
        "english_stop": {
          "type": "stop",
          "stopwords": "_english_"
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "description" : {
        "type": "text",
        "analyzer": "custom_analyzer"
      }
```

```
    }
  }
}
```

3. Create sample documents using the _bulk endpoint:

```
POST /product_catalog/_bulk
{ "index": { "_id": "1" } }
{ "description": "This is a great product! It works perfectly." }
{ "index": { "_id": "2" } }
{ "description": "An amazing gadget, with excellent features." }
```

Test

1. Analyze a sample text to verify the custom analyzer:

```
GET product_catalog/_analyze
{
  "analyzer" : "custom_analyzer",
  "text" : "i2can2RUN4the6MARATHON!"
}
```

```
// response
{
  "tokens": [
    {
      "token": "i",
      "start_offset": 0,
      "end_offset": 1,
      "type": "word",
      "position": 0
    },
    {
      "token": "can",
      "start_offset": 2,
      "end_offset": 5,
      "type": "word",
      "position": 1
    },
    {
      "token": "run",
      "start_offset": 6,
```

```
      "end_offset": 9,
      "type": "word",
      "position": 2
    },
    {
      "token": "marathon",
      "start_offset": 14,
      "end_offset": 22,
      "type": "word",
      "position": 4
    }
  ]
}
```

2. Search for documents to confirm they are indexed correctly:

```
GET /product_catalog/_search
{
  "query": {
    "match": {
      "description": "great product"
    }
  }
}
```

Considerations

- The custom tokenizer splits text on non-letter characters, ensuring that punctuation does not affect tokenization.
  - The `lowercase` tokenizer splits text on non-letter characters and turns uppercase characters into lowercase
- The `lowercase` filter normalizes text to lower case, providing case-insensitive searches.
- The `custom_stop` stopword filter removes common English stopwords, improving search relevance by ignoring less important words.

Clean-up (optional)

- Delete the `ecommerce_products` index:

```
DELETE /ecommerce_products
```

Documentation

- Bulk API
- Custom Analyzers
- Lowercase Tokenizer
- Lowercase Filter
- Stop word Filter

Example 4: Create a Custom Analyzer for E-commerce Product Data

Requirements

- Index e-commerce product data with fields such as `name`, `category`, `description`, and `sku`.
- Custom analyzer to normalize text for consistent search results, including handling special characters and case sensitivity.
- Use the `_bulk` endpoint to ingest multiple documents.
- Two example searches to verify that the custom analyzer handles both hyphenated and non-hyphenated queries.

Steps

1. **Define the Custom Analyzer:**
    - Set up the analyzer to lowercase text, remove special characters, and tokenize the content.

```
PUT /ecommerce_products
{
  "settings": {
    "analysis": {
      "char_filter": {
        "remove_special_chars": {
          "type": "pattern_replace",
          "pattern": "[^\\w\\s]",
          "replacement": ""
        }
      },
```

```
      "filter": {
        "my_lowercase": {
          "type": "lowercase"
        }
      },
      "analyzer": {
        "custom_analyzer": {
          "char_filter": ["remove_special_chars"],
          "tokenizer": "standard",
          "filter": ["my_lowercase"]
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "name": {
        "type": "text",
        "analyzer": "custom_analyzer"
      },
      "category": {
        "type": "keyword"
      },
      "description": {
        "type": "text",
        "analyzer": "custom_analyzer"
      },
      "sku": {
        "type": "keyword"
      }
    }
  }
}
```

2. **Index Sample Documents Using `_bulk` Endpoint:**

   - Use the `_bulk` endpoint to ingest multiple documents.

```
POST /ecommerce_products/_bulk
{ "index": { "_id": "1" } }
{ "name": "Choco-Lite Bar", "category": "Snacks", "description": "A
↪  light and crispy chocolate snack bar.", "sku": "SNACK-CHOCOLITE-001"
↪  }
```

```
{ "index": { "_id": "2" } }
{ "name": "Apple iPhone 12", "category": "Electronics", "description": "The latest iPhone model with advanced features.", "sku": "ELEC-IPH12-256GB" }
{ "index": { "_id": "3" } }
{ "name": "Samsung Galaxy S21", "category": "Electronics", "description": "A powerful smartphone with an impressive camera.", "sku": "ELEC-SG-S21" }
{ "index": { "_id": "4" } }
{ "name": "Nike Air Max 270", "category": "Footwear", "description": "Comfortable and stylish sneakers.", "sku": "FTWR-NIKE-AM270" }
```

Test

- **Query without Hyphen:**

```
GET /ecommerce_products/_search
{
  "query": {
    "match": {
      "name": "chocolite"
    }
  }
}
```

- **Query with Hyphen:**

```
GET /ecommerce_products/_search
{
  "query": {
    "match": {
      "name": "choco-lite"
    }
  }
}
```

Considerations

- The `pattern_replace` character filter removes non-alphanumeric characters (excluding whitespace) to normalize data for indexing and searching.

- The `lowercase` filter ensures case-insensitivity, providing consistent search results regardless of the case of the input.
- The use of the `_bulk` endpoint allows efficient indexing of multiple documents in a single request, which is especially useful for large datasets.

Documentation

- Custom Analyzer
- Pattern Replace Char Filter
- Lowercase Token Filter
- Standard Tokenizer
- Bulk API

4.3 Task: Define and use multi-fields with different data types and/or analyzers

Example 1: Creating multi-fields for product names in an e-commerce catalog

Requirements

- Define an index called `product_catalog`
- Define a field with a `text` type for full-text search.
- Include a `keyword` sub-field for exact matches.
- Add a custom analyzer to the text field to normalize the text.

Steps

1. **Open the Kibana Console** or use a REST client
2. Define the multi-fields in the index mappings

```
PUT /product_catalog
{
  "settings": {
    "analysis": {
      "analyzer": {
        "custom_analyzer": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "asciifolding"
          ]
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "product_name": {
        "type": "text",
        "analyzer": "custom_analyzer",
        "fields": {
          "keyword": {
            "type": "keyword",
```

```
            "ignore_above": 256
          }
        }
      }
    }
  }
}
```

3. Create sample documents using the `_bulk` endpoint:

```
POST /product_catalog/_bulk
{ "index": { "_id": "1" } }
{ "product_name": "Deluxe Toaster" }
{ "index": { "_id": "2" } }
{ "product_name": "Premium Coffee Maker" }
```

Test

1. Retrieve the index configuration to verify the custom analyzer and the sub-field:

```
GET product_catalog
```

2. Search for documents using the text field:

```
GET /product_catalog/_search
{
  "query": {
    "match": {
      "product_name": "deluxe"
    }
  }
}
```

3. Search for documents using the keyword sub-field:

```
GET /product_catalog/_search
{
  "query": {
    "term": {
      "product_name.keyword": "Deluxe Toaster"
    }
  }
}
```

Considerations

- The custom analyzer (`standard`) includes the lowercase filter for case-insensitive searches.
- The `keyword` sub-field allows for exact matches, which is useful for aggregations and sorting.

Clean-up (optional)

- Delete the Index

```
DELETE /product_catalog
```

Documentation

- Bulk API
- Custom Analyzers
- Multi-fields

Example 2: Creating a multi-field for a title with different analyzers

Requirements

- Create a mapping for a index named `myindex`
- The `title` field should have a sub-field for exact matching (`keyword`)
- The `title` field should have a sub-field for full-text search (`text`) with standard analyzer
- The `title` field should have a sub-field for full-text search (`text`) with english analyzer

Steps

1. **Open the Kibana Console** or use a REST client
2. Create an index with the desired mapping:

```
PUT /myindex
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "fields": {
          "exact": {
            "type": "keyword"
          },
          "std": {
            "type": "text",
            "analyzer": "standard"
          },
          "english": {
            "type": "text",
            "analyzer": "english"
          }
        }
      }
    }
  }
}
```

3. Add documents using the appropriate endpoint:

```
POST /myindex/_bulk
{ "index": { "_index": "myindex" } }
{ "title": "The Quick Brown Fox" }
{ "index": { "_index": "myindex" } }
{ "title": "The Quick Brown Fox Jumps" }
```

Test

- Verify the index was created with its associated multi-fields

```
GET myindex
```

- Use the _search API to verify that the multi-field is working correctly

```
GET /myindex/_search
{
  "query": {
    "match": {
      "title.exact": "The Quick Brown Fox"
    }
  }
}

GET /myindex/_search
{
  "query": {
    "match": {
      "title.std": "Quick Brown"
    }
  }
}

GET /myindex/_search
{
  "query": {
    "match": {
      "title.english": "Quick Brown"
    }
  }
}
```

Considerations

- The `title.exact` sub-field is used for exact matching.
- The `title.std` sub-field is used for full-text search with the standard analyzer.
- The `title.english` sub-field is used for full-text search with the English analyzer.

Clean-up (optional)

- Delete the Index

```
DELETE /myindex
```

Documentation

- Bulk API
- Multi-Field

Example 3: Creating multi-fields for analyzing text data

Requirements

- Create a mapping for a index named `text_data`
- Store the original text data in `content` for display purposes
- Analyze the text data for full-text search
- Analyze the text data for filtering and aggregations

Steps

1. **Open the Kibana Console** or use a REST client
2. Define the multi-fields in the index mapping

```
PUT /text_data
{
  "mappings": {
    "properties": {
      "content": {
        "type": "text",
        "fields": {
          "raw": {
            "type": "keyword"
          },
          "analyzed": {
            "type": "text",
            "analyzer": "english"
          },
          "ngram": {
            "type": "text",
            "analyzer": "ngram_analyzer"
          }
        }
      }
    }
  },
  "settings": {
    "analysis": {
      "analyzer": {
        "ngram_analyzer": {
          "tokenizer": "ngram_tokenizer"
        }
```

```
      },
      "tokenizer": {
        "ngram_tokenizer": {
          "type": "ngram",
          "min_gram": 2,
          "max_gram": 3
        }
      }
    }
  }
}
```

3. Index some documents using the text_data index:

```
POST /text_data/_bulk
{ "index": {} }
{ "content": "This is a sample text for analyzing." }
{ "index": {} }
{ "content": "Another example of text data." }
```

Test

1. Verify the index was created with its associated multi-fields

```
GET text_data
```

2. Test the multi-fields by querying and aggregating the data:

```
GET /text_data/_search
{
  "query": {
    "match": {
      "content.analyzed": "sample"
    }
  },
  "aggs": {
    "filter_agg": {
      "filter": {
        "term": {
          "content.ngram": "ex"
        }
      }
```

```
    }
  }
}
```

The output should show a single document in the search results matching the analyzed text and the aggregation results based on the ngram analysis.

The following:

```
GET /text_data/_search
{
  "query": {
    "match": {
      "content.ngram": "ex"
    }
  },
  "aggs": {
    "filter_agg": {
      "filter": {
        "term": {
          "content.ngram": "ex"
        }
      }
    }
  }
}
```

will show 2 documents as the search is looking for the substring “ex” which can be found in both documents, but only if you search against `content.ngram`.

```
// edited response
{
  ...
  "hits": {
    "total": {
      "value": 1,
      "relation": "eq"
    },
    "max_score": 0.7361701,
    "hits": [
      {
        "_index": "text_data",
        "_id": "qnqiBJEBRRh1FLFiJKsV",
```

```
        "_score": 0.7361701,
        "_source": {
          "content": "This is a sample text for analyzing."
        }
      }
    ]
  },
  "aggregations": {
    "filter_agg": {
      "doc_count": 1
    }
  }
}
```

Considerations

- The `content` field has multiple sub-fields: `raw` (keyword), `analyzed` (text with English analyzer), and `ngram` (text with ngram analyzer).
- The `raw` sub-field is used for storing the original text data without analysis.
- The `analyzed` sub-field is used for full-text search using the English analyzer.
- The `ngram` sub-field is used for filtering and aggregations based on ngram analysis.

Clean-up (optional)

- Delete the Index

```
DELETE text_data
```

Documentation

- Analyzers
- Multi-fields
- Ngram Tokenizer

4.4 Task: Use the Reindex API and Update By Query API to reindex and/or update documents

Example 1: Moving and updating product data to a new index with a new field

Requirements

- Reindex data from an existing index named `products_old` to a new index named `products_new`.
- During the reindexing process, add a new field named `stock_level` with a default value of **10** for each product.

Steps

1. **Open the Kibana Console** or use a REST client
2. Create the indices (notice that they both look identical){target="_blank"}

```
PUT /products_old
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
    "properties": {
      "product_id": {
        "type": "keyword"
      },
      "name": {
        "type": "text"
      },
      "description": {
        "type": "text"
      },
      "price": {
        "type": "double"
      },
      "availability_status": {
        "type": "boolean"
      }
    }
```

```
  }
}
```

```
PUT /products_new
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
    "properties": {
      "product_id": {
        "type": "keyword"
      },
      "name": {
        "type": "text"
      },
      "description": {
        "type": "text"
      },
      "price": {
        "type": "double"
      },
      "availability_status": {
        "type": "boolean"
      }
    }
  }
}
```

3. Add products to `products_old`

```
POST /products_old/_bulk
{ "index": { "_index": "products_old", "_id": "1" } }
{ "product_id": "1", "name": "Wireless Mouse", "description": "A
 ↪  high-quality wireless mouse with ergonomic design.", "price": 29.99,
 ↪  "availability_status": true }
{ "index": { "_index": "products_old", "_id": "2" } }
{ "product_id": "2", "name": "Gaming Keyboard", "description":
 ↪  "Mechanical gaming keyboard with customizable RGB lighting.",
 ↪  "price": 79.99, "availability_status": true }
{ "index": { "_index": "products_old", "_id": "3" } }
```

```
{ "product_id": "3", "name": "USB-C Hub", "description": "A versatile
 ↪  USB-C hub with multiple ports.", "price": 49.99,
 ↪  "availability_status": true }
```

4. Use the **Reindex API** with a script to update documents during the copy process:

```
POST /_reindex
{
  "source": {
    "index": "products_old"
  },
  "dest": {
    "index": "products_new"
  },
  "script": {
    "source": "ctx._source.stock_level = 10"
  }
}
```

5. Wait for the reindexing or update operation to complete.

Test

1. Verify that the documents from `products_old` do not contain `stock_level`

```
GET /products_old/_search
```

```
// edited response
{
 ...
     "hits": [
       {
         "_index": "products_old",
         "_id": "1",
         "_score": 1,
         "_source": {
           "product_id": "1",
           "name": "Wireless Mouse",
           "description": "A high-quality wireless mouse with ergonomic
            ↪  design.",
           "price": 29.99,
           "availability_status": true
```

```
        }
      },
      {
        "_index": "products_old",
        "_id": "2",
        "_score": 1,
        "_source": {
          "product_id": "2",
          "name": "Gaming Keyboard",
          "description": "Mechanical gaming keyboard with customizable
          ↪  RGB lighting.",
          "price": 79.99,
          "availability_status": true
        }
      },
      {
        "_index": "products_old",
        "_id": "3",
        "_score": 1,
        "_source": {
          "product_id": "3",
          "name": "USB-C Hub",
          "description": "A versatile USB-C hub with multiple ports.",
          "price": 49.99,
          "availability_status": true
        }
      }
    ]
  }
}
```

2. Verify that the data is successfully migrated to the `products_new` index with the addition of `stock_level`

```
GET /products_new/_search
```

```
// edited response
{
  ...
    "hits": [
      {
        "_index": "products_new",
```

```
        "_id": "1",
        "_score": 1,
        "_source": {
          "availability_status": true,
          "price": 29.99,
          "product_id": "1",
          "stock_level": 10,
          "name": "Wireless Mouse",
          "description": "A high-quality wireless mouse with ergonomic
            ↪ design."
        }
      },
      {
        "_index": "products_new",
        "_id": "2",
        "_score": 1,
        "_source": {
          "availability_status": true,
          "price": 79.99,
          "product_id": "2",
          "stock_level": 10,
          "name": "Gaming Keyboard",
          "description": "Mechanical gaming keyboard with customizable
            ↪ RGB lighting."
        }
      },
      {
        "_index": "products_new",
        "_id": "3",
        "_score": 1,
        "_source": {
          "availability_status": true,
          "price": 49.99,
          "product_id": "3",
          "stock_level": 10,
          "name": "USB-C Hub",
          "description": "A versatile USB-C hub with multiple ports."
        }
      }
    ]
  }
}
```

Considerations

- The **Reindex API** with a script allows copying data and applying transformations during the process.

Clean-up (optional)

- Delete the two indices

```
DELETE products_old
DELETE products_new
```

Documentation

- Reindex API
- Update By Query API

Example 2: Reindexing and updating product data

Requirements

- Reindex data from an existing index named `products_old` to a new index named `products_new`.
- Both indices have the following fields:
  - `name` (`text`)
  - `price` (`float`)
  - `inventory_count` (`integer`)
- The `products_new` index has an additional `boolean` field called `in_stock`
- In `products_new`, update the `in_stock` field for products with a low inventory count (less than 10 items)

Steps

1. **Open the Kibana Console** or use a REST client
2. Create the old index with some sample data:

```
PUT /products_old
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "price": {
        "type": "float"
      },
      "inventory_count": {
        "type": "integer"
      }
    }
  }
}
```

```
POST /products_old/_bulk
{ "index": {} }
{ "name": "Product A", "price": 19.99, "inventory_count": 10 }
{ "index": {} }
{ "name": "Product B", "price": 29.99, "inventory_count": 5 }
```

```
{ "index": {} }
{ "name": "Product C", "price": 39.99, "inventory_count": 20 }
```

3. Create the new index with an updated mapping:

```
PUT /products_new
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "price": {
        "type": "float"
      },
      "inventory_count": {
        "type": "integer"
      },
      "in_stock": {
        "type": "boolean"
      }
    }
  }
}
```

4. Reindex the data from the old index to the new index. This updates the `in_stock` field as it migrates the content.

```
POST /_reindex
{
  "source": {
    "index": "products_old"
  },
  "dest": {
    "index": "products_new"
  },
  "script": {
    "source": """
      if (ctx._source.inventory_count < 10) {
        ctx._source.in_stock = false;
      } else {
        ctx._source.in_stock = true;
      }
```

```
    """
  }
}
```

5. You also update the `in_stock` field for products with low inventory after the content is reindexed/migrated.

```
POST /products_new/_update_by_query
{
  "script": {
    "source": "ctx._source.in_stock = false"
  },
  "query": {
    "range": {
      "inventory_count": {
        "lt": 10
      }
    }
  }
}
```

Test

1. Search the new index to verify the reindexed data and updated `in_stock` field

```
GET /products_new/_search
```

```
// edited response
{
  ...
    "hits": [
      {
        "_index": "products_new",
        "_id": "rHqtBJEBRRh1FLFi_quh",
        "_score": 1,
        "_source": {
          "price": 19.99,
          "inventory_count": 10,
          "name": "Product A",
          "in_stock": true
        }
      },
```

```
      {
        "_index": "products_new",
        "_id": "rXqtBJEBRRh1FLFi_qui",
        "_score": 1,
        "_source": {
          "price": 29.99,
          "inventory_count": 5,
          "name": "Product B",
          "in_stock": false
        }
      },
      {
        "_index": "products_new",
        "_id": "rnqtBJEBRRh1FLFi_qui",
        "_score": 1,
        "_source": {
          "price": 39.99,
          "inventory_count": 20,
          "name": "Product C",
          "in_stock": true
        }
      }
    ]
  }
}
```

The response should show the reindexed products with `in_stock` set correctly based on the inventory count.

2. Search `products_old` to verify the original data and the absence of `in_stock`

```
GET /products_old/_search
```

```
// edited response
{
  ...
    "hits": [
      {
        "_index": "products_old",
        "_id": "rHqtBJEBRRh1FLFi_quh",
        "_score": 1,
        "_source": {
          "name": "Product A",
```

```
          "price": 19.99,
          "inventory_count": 10
        }
      },
      {
        "_index": "products_old",
        "_id": "rXqtBJEBRRh1FLFi_qui",
        "_score": 1,
        "_source": {
          "name": "Product B",
          "price": 29.99,
          "inventory_count": 5
        }
      },
      {
        "_index": "products_old",
        "_id": "rnqtBJEBRRh1FLFi_qui",
        "_score": 1,
        "_source": {
          "name": "Product C",
          "price": 39.99,
          "inventory_count": 20
        }
      }
    ]
  }
}
```

Considerations

- The ***Reindex API*** is used to copy data from the old index to the new index while applying a script to set the "in_stock" field based on the inventory count.
- The ***Update By Query API*** is used to update the `in_stock` field for products with an inventory count lower than **10**.

Clean-up (optional)

- Delete the two indices

```
DELETE products_old
DELETE products_new
```

Documentation

- Reindex API
- Update By Query API
- Scripting

Example 3: Reindexing documents from an old product catalog to a new one with updated mappings and updating prices in the new catalog

Requirements

- Create the `products_old` index and add sample products.
- Create the `products_new` index using the `products_old` mapping.
- Reindex documents from `products_old` to `products_new`.
  - Increase the price of all products in `products_new` by 10%.

Steps

1. Create the `products_old` index and add sample products

```
PUT /products_old
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
    "properties": {
      "product_id": {
        "type": "keyword"
      },
      "name": {
        "type": "text"
      },
      "description": {
        "type": "text"
      },
      "price": {
        "type": "double"
      },
      "availability_status": {
        "type": "boolean"
      }
    }
  }
}

POST /products_old/_bulk
```

```
{ "index": { "_index": "products_old", "_id": "1" } }
{ "product_id": "1", "name": "Wireless Mouse", "description": "A high-quality wireless mouse with ergonomic design.", "price": 29.99, "availability_status": true }
{ "index": { "_index": "products_old", "_id": "2" } }
{ "product_id": "2", "name": "Gaming Keyboard", "description": "Mechanical gaming keyboard with customizable RGB lighting.", "price": 79.99, "availability_status": true }
{ "index": { "_index": "products_old", "_id": "3" } }
{ "product_id": "3", "name": "USB-C Hub", "description": "A versatile USB-C hub with multiple ports.", "price": 49.99, "availability_status": true }
```

2. Create the new index with updated mappings

   - Define the new index `products_new` with the desired mappings.

```
PUT /products_new
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
    "properties": {
      "product_id": {
        "type": "keyword"
      },
      "name": {
        "type": "text"
      },
      "description": {
        "type": "text"
      },
      "price": {
        "type": "double"
      },
      "availability_status": {
        "type": "boolean"
      }
    }
  }
```

```
}
```

3. Reindex Documents from `products_old` to `products_new` while updating `price`

```
POST _reindex
{
  "source": {
    "index": "products_old"
  },
  "dest": {
    "index": "products_new"
  },
  "script": {
    "source": "ctx._source.price *= 1.1;"
  }
}
```

4. **OR** Migrate the content and then update `price` in the new index using the **Update By Query API** to increase the price of all products in `products_new` by 10%.

```
POST _reindex
{
  "source": {
    "index": "products_old"
  },
  "dest": {
    "index": "products_new"
  }
}
```

```
POST /products_new/_update_by_query
{
  "script": {
    "source": "ctx._source.price *= 1.10",
    "lang": "painless"
  },
  "query": {
    "match_all": {}
  }
}
```

Test

1. Verify the reindexing

```
GET /products_old/_count
GET /products_new/_count
```

```
// responses for both indices
# GET /products_old/_count 200 OK
{
  "count": 3,
  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped": 0,
    "failed": 0
  }
}
# GET /products_new/_count 200 OK
{
  "count": 3,
  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped": 0,
    "failed": 0
  }
}
```

2. Verify the price update

```
GET /products_old,products_new/_search
{
  "query": {
    "match_all": {}
  },
  "_source": [
    "price"
    ]
}
```

```
// edited response
{
```

```
...
  "hits": [
    {
      "_index": "products_new",
      "_id": "1",
      "_score": 1,
      "_source": {
        "price": 32.989000000000004
      }
    },
    {
      "_index": "products_new",
      "_id": "2",
      "_score": 1,
      "_source": {
        "price": 87.989
      }
    },
    {
      "_index": "products_new",
      "_id": "3",
      "_score": 1,
      "_source": {
        "price": 54.989000000000004
      }
    },
    {
      "_index": "products_old",
      "_id": "1",
      "_score": 1,
      "_source": {
        "price": 29.99
      }
    },
    {
      "_index": "products_old",
      "_id": "2",
      "_score": 1,
      "_source": {
        "price": 79.99
      }
    },
```

```
      {
        "_index": "products_old",
        "_id": "3",
        "_score": 1,
        "_source": {
          "price": 49.99
        }
      }
    ]
  }
}
```

Considerations

- **Mappings Update**: Ensure the new index `products_new` has the updated mappings to accommodate any changes in the document structure.
- **Price Update Script**: The script in the **Update By Query API** uses the `painless` language to increase the price by 10%. This is a simple and efficient way to update document fields.

Clean-up (optional)

- Delete the indices

```
DELETE /products_old
DELETE /products_new
```

Documentation

- Index Settings
- Painless Scripting Language
- Reindex API
- Update By Query API

4.5 Task: Define and use an ingest pipeline that satisfies a given set of requirements, including the use of Painless to modify documents

Example 1: Create an ingest pipeline for enriching and modifying product data in an e-commerce catalog

Requirements

- Create an ingest pipeline named `product_pipeline` to process incoming documents.
- Apply a Painless script to modify `price` to add **10%** to the price
- Enrich the data by adding the ingest time to a `timestamp` field
- Create a `product_catalog` index

Notes: the use of the `ctx` object which represents a single document being processed. When **updating** a field (meaning the doc already exists in the index) you use the following form:

```
ctx._source.[field name]
```

vs. directly accessing the field in question prior to it being indexed:

```
ctx.[field name]
```

Steps

1. **Open the Kibana Console** or use a REST client
2. Define the ingest pipeline with a **Painless** script and additional processors:

```
PUT /_ingest/pipeline/product_pipeline
{
  "processors": [
    {
      "script": {
        "lang": "painless",
        "source": """
          if (ctx.price != null) {
            ctx.price *= 1.1;
          }
        """
      }
```

```
    },
    {
      "set": {
        "field": "timestamp",
        "value": "{{_ingest.timestamp}}"
      }
    }
  ]
}
```

3. Create the `product_catalog` index

```
PUT /product_catalog
{
  "mappings": {
    "properties": {
      "product_id": {
        "type": "keyword"
      },
      "name": {
        "type": "text"
      },
      "description": {
        "type": "text"
      },
      "price": {
        "type": "double"
      },
      "timestamp": {
        "type": "date"
      }
    }
  }
}
```

4. Index documents using the ingest pipeline

```
POST /product_catalog/_bulk?pipeline=product_pipeline
{ "index": { "_id": "1" } }
{ "product_id": "p001", "name": "Product 1", "description": "Description
 ↪  of product 1", "price": 20.0 }
{ "index": { "_id": "2" } }
{ "product_id": "p002", "name": "Product 2", "description": "Description
 ↪  of product 2", "price": 30.0 }
```

Test

1. Verify the ingest pipeline configuration:

```
GET /_ingest/pipeline/product_pipeline
```

2. Search the indexed documents to ensure the modifications have been applied:

```
GET /product_catalog/_search
```

Considerations

- The **Painless** script modifies the `price` field to contain a 10% higher price
- The `set` processor adds a `timestamp` to each document to track when it was ingested.
- The inkjest pipeline processes all incoming documents to maintain data consistency.

Clean-up (optional)

- Delete the index

```
DELETE product_catalog
```

- Delete the pipeline

```
DELETE _ingest/pipeline/product_pipeline
```

Documentation

- Bulk API
- Ingest Node Pipelines
- Painless Scripting Language

Example 2: Creating an ingest pipeline to extract and transform data for a logging index

This example creates another ingest pipeline, but this time adds it directly into the index definition.

This is also an example of how helpful it is to know more about scripting in Elasticsearch. The examples may or may not be trivial/complex, but an understanding of how to write script is required.

Requirements

- Create an ingest pipeline named `logging-pipeline`
- Extract from the log message:
  - the log level (`DEBUG`, `INFO`, `WARNING`, `ERROR`)
  - the log timestamp in ISO format
- Add a new field `log_level_tag` with a value based on the log level (e.g. `DEBUG` -> `DEBUG_LOG`).
- Add a new field `log_timestamp_in_seconds` with the timestamp in seconds.
- Create a `logging-index` index
  - Declare the ingest pipeline as the defaultin the `logging-index` index `settings`

Steps

1. **Open the Kibana Console** or use a REST client
2. Create an ingest pipeline:

```
PUT /_ingest/pipeline/logging-pipeline
{
  "description": "Extract and transform log data",
  "processors": [
    {
      "grok": {
        "field": "message",
        "patterns": ["%{LOGLEVEL:log_level}
        ↪  %{TIMESTAMP_ISO8601:log_timestamp} %{GREEDYDATA:message}"]
      }
    },
    {
      "script": {
```

```
        "source": """
          ctx.log_level_tag = ctx.log_level.toUpperCase() + '_LOG';
          ctx.log_timestamp_in_seconds =
 ↪  ZonedDateTime.parse(ctx.log_timestamp).toEpochSecond();
        """,
        "lang": "painless"
      }
    }
  ]
}
```

3. Create an index with the ingest pipeline:

```
PUT /logging-index
{
  "mappings": {
    "properties": {
      "message": {
        "type": "text"
      },
      "log_level": {
        "type": "keyword"
      },
      "log_timestamp": {
        "type": "date"
      },
      "log_level_tag": {
        "type": "keyword"
      },
      "log_timestamp_in_seconds": {
        "type": "long"
      }
    }
  },
  "settings": {
    "index": {
      "default_pipeline": "logging-pipeline"
    }
  }
}
```

4. Add documents to the index:

```
POST /logging-index/_bulk
{ "index": { "_index": "logging-index" } }
{ "message": "DEBUG 2022-05-25T14:30:00.000Z This is a debug message" }
{ "index": { "_index": "logging-index" } }
{ "message": "INFO 2022-05-25T14:30:00.000Z This is an info message" }
```

Test

- Verify that the documents have been processed correctly:

```
GET /logging-index/_search
```

```
// edited response
{
  ...
    "hits": [
      {
        "_index": "logging-index",
        "_id": "uXpCBpEBRRh1FLFiQ6s4",
        "_score": 1,
        "_source": {
          "log_level": "DEBUG",
          "log_timestamp": "2022-05-25T14:30:00.000Z",
          "log_level_tag": "DEBUG_LOG",
          "message": "This is a debug message",
          "log_timestamp_in_seconds": 1653489000
        }
      },
      {
        "_index": "logging-index",
        "_id": "unpCBpEBRRh1FLFiQ6s4",
        "_score": 1,
        "_source": {
          "log_level": "INFO",
          "log_timestamp": "2022-05-25T14:30:00.000Z",
          "log_level_tag": "INFO_LOG",
          "message": "This is an info message",
          "log_timestamp_in_seconds": 1653489000
        }
      }
    ]
```

```
  }
}
```

Considerations

- The ingest pipeline uses the **Grok** processor to extract the log level and timestamp from the log message.
- The **Painless** script processor is used to transform the log level and timestamp into new fields.

Clean-up (optional)

- Delete the index

```
DELETE logging-index
```

- Delete the pipeline

```
DELETE _ingest/pipeline/logging-pipeline
```

Documentation

- Ingest Node Pipelines
- Painless Scripting Language

Example 3: Creating an ingest pipeline for product data

Requirements

- Create an index mapping for `products` with fields like `name`, `price`, `category`, `description`, `discounted_price`.
- Preprocess incoming product data using an ingest pipeline called `product_pipeline`:
  - Lowercase the `name` and `category` fields
  - Remove HTML tags from the `description` field
  - Calculate a `discounted_price` field based on the `price` field and a discount percentage stored in a pipeline variable

Steps

1. **Open the Kibana Console** or use a REST client
2. Define the ingest pipeline:

```
PUT _ingest/pipeline/product_pipeline
{
  "processors": [
    {
      "lowercase": {
        "field": "name"
      },
      "html_strip": {
        "field": "description"
      },
      "script": {
        "source": "double discount = 0.1; ctx.discounted_price =
        ↪  ctx.price * (1 - discount);"
      }
    },
    {
      "lowercase": {
        "field": "category"
      }
    }
  ]
}
```

3. Index a sample document using the ingest pipeline:

```
PUT /products/_doc/1?pipeline=product_pipeline
{
  "name": "Product A",
  "price": 99.99,
  "category": "Electronics",
  "description": "A <b>high-quality</b> product for running
   ↪  enthusiasts."
}
```

Test

1. Search the `products` index and verify that the document has been processed by the ingest pipeline:

```
GET /products/_search
```

```
// edited response
{
  ...
    "hits": [
      {
        "_index": "products",
        "_id": "1",
        "_score": 1,
        "_source": {
          "name": "product a",
          "description": "A high-quality product for running
           ↪  enthusiasts.",
          "category": "electronics",
          "price": 99.99,
          "discounted_price": 89.991
        }
      }
    ]
  }
}
```

Considerations

- The ingest pipeline is defined with a list of processors that perform specific operations on incoming documents.
- The `lowercase` processor lowercases the `name` and `category` fields.
- The `html_strip` processor removes HTML tags from `description`
- The `script` processor uses the Painless scripting language to calculate the `discounted_price` field based on the `price` field and a discount percentage variable.

Clean-up (optional)

- Delete the index

```
DELETE products
```

- Delete the pipeline

```
DELETE _ingest/pipeline/product_pipeline
```

Documentation

- Ingest Node
- Ingest Pipelines
- Ingest Processors
- Painless Scripting Language

Example 4: Merge content from two indices into a third index

Requirements

The `movie` index has content that looks like this:

```
{
  "movie_id": 1,
  "title": "The Adventure Begins",
  "release_year": 2021,
  "genre_code": "ACT"
}
```

The `genre` index has content that looks like this:

```
{
  "genre_code": "ACT",
  "description": "Action - Movies with high energy and lots of physical
   ↪  activity"
}
```

Merge `movie` and `genre` into a third index called `movie_with_genre` that includes the `genre.description` in each `movie` record:

```
{
  "movie_id": 1,
  "title": "The Adventure Begins",
  "release_year": 2021,
  "genre_code": "ACT",
  "genre_description": "Action - Movies with high energy and lots of physical
   ↪  activity"
}
```

Steps

In order to merge two or more indices into a third index you will need to create an ingest pipeline that uses an index management enrich policy.

1. Create an enrich policy that contains the index with the additional content to be used

2. Execute the policy to create an enrich index as a temporary location for the enrich content
3. Create an ingest pipeline that points to the enrich policy and the input index that will be merged with the enrich index

FROM THE KIBANA UI

1. **Open the Kibana Console** or use a REST client

   - Create the `movie` index with sample documents

```
PUT /movie
{
  "mappings": {
    "properties": {
      "movie_id": { "type": "integer" },
      "title": { "type": "text" },
      "release_year": { "type": "integer" },
      "genre_code": { "type": "keyword" }
    }
  }
}

POST /movie/_bulk
{ "index": { "_id": 1 } }
{ "movie_id": 1, "title": "The Adventure Begins", "release_year": 2021, "genre_code": "ACT" }
{ "index": { "_id": 2 } }
{ "movie_id": 2, "title": "Drama Unfolds", "release_year": 2019, "genre_code": "DRM" }
{ "index": { "_id": 3 } }
{ "movie_id": 3, "title": "Comedy Night", "release_year": 2020, "genre_code": "COM" }
{ "index": { "_id": 4 } }
{ "movie_id": 4, "title": "Epic Adventure", "release_year": 2022, "genre_code": "ACT" }
{ "index": { "_id": 5 } }
{ "movie_id": 5, "title": "Tragic Tale", "release_year": 2018, "genre_code": "DRM" }
```

   - Create the `genre` index with sample documents

```
PUT /genre
{
```

```
  "mappings": {
    "properties": {
      "genre_code": { "type": "keyword" },
      "description": { "type": "text" }
    }
  }
}

POST /genre/_bulk
{ "index": { "_id": "ACT" } }
{ "genre_code": "ACT", "description": "Action - Movies with high energy and lots of physical activity" }
{ "index": { "_id": "DRM" } }
{ "genre_code": "DRM", "description": "Drama - Movies with serious, emotional, and often realistic stories" }
{ "index": { "_id": "COM" } }
{ "genre_code": "COM", "description": "Comedy - Movies designed to make the audience laugh" }
```

- Optionally, create the `movie_with_genre` index

```
PUT /movie_with_genre
{
  "mappings": {
    "properties": {
      "movie_id": { "type": "integer" },
      "title": { "type": "text" },
      "release_year": { "type": "integer" },
      "genre_code": { "type": "keyword" },
      "genre_description": { "type": "text" }
    }
  }
}
```

2. From the Kibana dashboard: **Home > Management > Index Management**
3. Press **Add an Enrich Policy**

   Configuration

   - **Policy Name:** movie-genre-policy
   - **Policy Type:** Match
   - **Source Indices:** genre

Next: **Field Selection**

- **Match field:** genre_code
- **Enrich field:** description

Next: **Create**

Press **Create and Execute** (if everything looks correct)

4. **Home > Management > Data > Ingest > Ingest Pipelines**

- Press: Create Pipeline > New Pipeline

  Create Pipeline

  - **Name:** genre_ingest_pipeline
  - **Press:** Add Your First Processor > Add a Processor
    - Add Processor
      - **Processor:** Enrich
      - **Field:** genre_code (from the `movie` index)
      - **Policy name:** movie-genre-policy
      - **Target field:** genre_description (from `movie_with_genre` index)
    - Press **Add Processor**
  - Press: **Test Document: Add Documents**

  Enter:

```
[
  {
    "_index": "movie",
    "_source": {
      "movie_id": 1,
      "title": "The Adventure Begins",
      "release_year": 2021,
      "genre_code": "ACT"
    }
  }
]
```

Press: **Run the Pipeline**

If the information entered is correct the response will be:

```
{
  "docs": [
    {
      "doc": {
        "_index": "movie",
        "_version": "-3",
        "_id": "_id",
        "_source": {
          "release_year": 2021,
          "genre_description": {
            "description": "Action - Movies with high energy and lots of
             ↪ physical activity",
            "genre_code": "ACT"
          },
          "movie_id": 1,
          "title": "The Adventure Begins",
          "genre_code": "ACT"
        },
        "_ingest": {
          "timestamp": "2024-08-04T17:18:50.159798109Z"
        }
      }
    }
  ]
}
```

Which is wrong as we just want the genre `description` field and not both the `genre_code` and `description`. The answer is given in the JSON below.

Press the X in the top right hand corner of the panel to close the panel (not the browser).

Press: **Create Pipeline** (when thhe side panel opens press **Close**)

WTF? Not sure why the enrich pipeline does that, but it needs to be corrected.

FROM THE KIBANA CONSOLE

1. **Open the Kibana Console** or use a REST client
   - Create the movie index with sample documents

```
PUT /movie
{
  "mappings": {
```

```
    "properties": {
      "movie_id": { "type": "integer" },
      "title": { "type": "text" },
      "release_year": { "type": "integer" },
      "genre_code": { "type": "keyword" }
    }
  }
}

POST /movie/_bulk
{ "index": { "_id": 1 } }
{ "movie_id": 1, "title": "The Adventure Begins", "release_year": 2021,
 ↪  "genre_code": "ACT" }
{ "index": { "_id": 2 } }
{ "movie_id": 2, "title": "Drama Unfolds", "release_year": 2019,
 ↪  "genre_code": "DRM" }
{ "index": { "_id": 3 } }
{ "movie_id": 3, "title": "Comedy Night", "release_year": 2020,
 ↪  "genre_code": "COM" }
{ "index": { "_id": 4 } }
{ "movie_id": 4, "title": "Epic Adventure", "release_year": 2022,
 ↪  "genre_code": "ACT" }
{ "index": { "_id": 5 } }
{ "movie_id": 5, "title": "Tragic Tale", "release_year": 2018,
 ↪  "genre_code": "DRM" }
```

- Create the genre index with sample documents

```
PUT /genre
{
  "mappings": {
    "properties": {
      "genre_code": { "type": "keyword" },
      "description": { "type": "text" }
    }
  }
}

POST /genre/_bulk
{ "index": { "_id": "ACT" } }
{ "genre_code": "ACT", "description": "Action - Movies with high energy
 ↪  and lots of physical activity" }
```

```
{ "index": { "_id": "DRM" } }
{ "genre_code": "DRM", "description": "Drama - Movies with serious, emotional, and often realistic stories" }
{ "index": { "_id": "COM" } }
{ "genre_code": "COM", "description": "Comedy - Movies designed to make the audience laugh" }
```

- Optionally, create the `movie_with_genre` index

```
PUT /movie_with_genre
{
  "mappings": {
    "properties": {
      "movie_id": { "type": "integer" },
      "title": { "type": "text" },
      "release_year": { "type": "integer" },
      "genre_code": { "type": "keyword" },
      "genre_description": { "type": "text" }
    }
  }
}
```

2. Create an enrich policy

```
PUT /_enrich/policy/movie-genre-policy
{
  "match": {
    "indices": "genre",
    "match_field": "genre_code",
    "enrich_fields": ["description"]
  }
}
```

3. Execute the enrich policy

```
PUT _enrich/policy/movie-genre-policy/_execute
```

4. Define the ingest pipeline that will merge the content from `genre` into `movie_with_genre`. Notice the use of a temporary field as the `genre` content is being copied in its entirety into the new index. To correct that, we copy it into a temp field and then delete the temp field.

```
PUT _ingest/pipeline/movie_genre_pipeline
{
  "processors": [
    {
      "enrich": {
        "policy_name": "movie-genre-policy",
        "field": "genre_code",
        "target_field": "enriched_data",
        "max_matches": "1"
      }
    },
    {
      "script": {
        "source": """
          if (ctx.enriched_data != null && ctx.enriched_data.description
 ↪  != null) {
            ctx.genre_description = ctx.enriched_data.description;
          }
          ctx.remove("enriched_data");
        """
      }
    }
  ]
}
```

5. Reindex `movie` into `movie_with_genre`

```
POST _reindex
{
  "source": {
    "index": "movie"
  },
  "dest": {
    "index": "movie_with_genre",
    "pipeline": "movie_genre_pipeline"
  }
}
```

Test

1. Validate the creation of the movie index

```
GET movie/_search
```

2. Validate the creation of the genre index

```
GET genre/_search
```

3. Validate the creation of the enrich policy

```
GET _enrich/policy/movie-genre-policy
```

4. Validate the creation of the ingest pipeline

```
GET _ingest/pipeline/movie_genre_pipeline
```

5. Simulate the use of the ingest pipeline

```
GET _ingest/pipeline/movie_genre_pipeline/_simulate
{
  "docs": [
    {
      "_index": "movie",
      "_source": {
        "movie_id": 1,
        "title": "The Adventure Begins",
        "release_year": 2021,
        "genre_code": "ACT"
      }
    }
  ]
}
```

6. Validate the `genre_description` in `movie_with_genre`

```
GET movie_with_genre/_search
{
  "query": {
    "match_all": {}
  },
  "_source": [ "genre_code", "genre_description" ]
}
```

```
// edited response
{
  ...
```

```
 "hits": [
   {
     "_index": "movie_with_genre",
     "_id": "1",
     "_score": 1,
     "_source": {
       "genre_description": "Action - Movies with high energy and lots
        ↪  of physical activity",
       "genre_code": "ACT"
     }
   },
   {
     "_index": "movie_with_genre",
     "_id": "2",
     "_score": 1,
     "_source": {
       "genre_description": "Drama - Movies with serious, emotional, and
        ↪  often realistic stories",
       "genre_code": "DRM"
     }
   },
   {
     "_index": "movie_with_genre",
     "_id": "3",
     "_score": 1,
     "_source": {
       "genre_description": "Comedy - Movies designed to make the
        ↪  audience laugh",
       "genre_code": "COM"
     }
   },
   ...
 ]
  }
}
```

Considerations

- The Painless script calculates a 10% discount on the price.
- Runtime fields are defined in the index mappings and can be used for querying and aggregations without being stored in the index.

Clean-up (optional)

- Delete the final index

```
DELETE movie_with_genre
```

- Delete the ingest pipeline and the enrich policy

```
DELETE _ingest/pipeline/movie_genre_pipeline
DELETE _enrich/policy/movie-genre-policy
```

- Delete the movie and genre indices

```
DELETE movie
DELETE genre
```

Documentation

- Enrich Processor Documentation
- Enrich Policy Management
- Ingest Node Overview
- Script Processor
- Reindex API
- Update By Query API

4.6 Task: Define runtime fields to retrieve custom values using Painless scripting

Example 1: Creating a runtime field for discounted prices in a product catalog

Requirements

- Create a mapping for the `product_catalog` index
  - Include runtime field `discounted_price` to calculate a discount on product prices.
  - Apply a **Painless** script to dynamically compute the discounted price.
  - Ensure the runtime field is available for queries and aggregations.

Steps

1. **Open the Kibana Console** or use a REST client
2. Define the index with appropriate mappings:

```
PUT /product_catalog
{
  "mappings": {
    "properties": {
      "product_id": {
        "type": "keyword"
      },
      "name": {
        "type": "text"
      },
      "description": {
        "type": "text"
      },
      "price": {
        "type": "double"
      }
    },
    "runtime": {
      "discounted_price": {
        "type": "double",
        "script": {
          "source": """
            if (doc['price'].size() != 0) {
```

```
              emit(doc['price'].value * 0.9);
            } else {
              emit(Double.NaN);
            }
          """
        }
      }
    }
  }
}
```

3. Index sample documents using the _bulk endpoint:

```
POST /product_catalog/_bulk
{ "index": { "_id": "1" } }
{ "product_id": "p001", "name": "Product 1", "description": "Description of product 1", "price": 20.0 }
{ "index": { "_id": "2" } }
{ "product_id": "p002", "name": "Product 2", "description": "Description of product 2", "price": 30.0 }
```

Test

1. Search the indexed documents and retrieve the runtime field

```
GET /product_catalog/_search
{
  "_source": ["name", "price"],
  "fields": ["discounted_price"],
  "query": {
    "match_all": {}
  }
}
```

```
// edited response
{
  ...
    "hits": [
      {
        "_index": "product_catalog",
        "_id": "1",
        "_score": 1,
```

```
        "_source": {
          "name": "Product 1",
          "price": 20
        },
        "fields": {
          "discounted_price": [
            18
          ]
        }
      },
      {
        "_index": "product_catalog",
        "_id": "2",
        "_score": 1,
        "_source": {
          "name": "Product 2",
          "price": 30
        },
        "fields": {
          "discounted_price": [
            27
          ]
        }
      }
    ]
  }
}
```

2. Verify the discounted price in the search results

```
GET /product_catalog/_search
{
  "query": {
    "match_all": {}
  },
  "script_fields": {
    "discounted_price": {
      "script": {
        "source": "doc['price'].value * 0.9"
      }
    }
  }
```

```
}
```

Considerations

- The Painless script calculates a 10% discount on the price.
- Runtime fields are defined in the index mappings and can be used for querying and aggregations without being stored in the index.

Clean-up (optional)

- Delete the index

```
DELETE product_catalog
```

Documentation

- Bulk API
- Painless Scripting Language
- Runtime Fields
- Scripts Fields

Example 2: Create a runtime field to extract the domain from a URL

Requirements

- Create a mapping to the `myindex` index
  - Define a field called `url`
- Extract the domain from a URL field using Painless scripting to define a runtime field named `domain`.

Steps

1. **Open the Kibana Console** or use a REST client
2. Create an index with a URL field:

```
PUT /myindex
{
  "mappings": {
    "properties": {
      "url": {
        "type": "keyword"
      }
    }
  }
}
```

3. Define a runtime field to extract the domain:

```
PUT myindex
{
  "mappings": {
    "properties": {
      "url": {
        "type": "keyword"
      }
    },
    "runtime": {
      "domain": {
        "type": "keyword",
        "script": {
          "source": """
          // https://xyz.domain.com/stuff/stuff
```

```
        String domain =
↪   grok("%{URIPROTO}://(?:%{USER}(?::[^@]*)?@)?(?:%{URIHOST:domain})?(?:%{URIPATHPARAM}
        if (domain != null) emit(domain);
        else emit("grok failed");
      """
        }
      }
    }
  }
}
```

4. Add documents to the index:

```
POST /myindex/_bulk
{ "index": { "_index": "myindex" } }
{ "url": "https://www.example.com/path/to/page" }
{ "index": { "_index": "myindex" } }
{ "url": "http://sub.example.com/other/page" }
```

Test

- Verify that the runtime field is working correctly:

```
GET /myindex/_search
{
  "query": {
    "match_all": {}
  },
  "fields": ["domain"]
}
```

Considerations

- The runtime field uses Painless scripting to extract the domain from the URL field.
- The script splits the URL into components and returns the domain (including the sub-domain. Removing it involves ugly logic).

Clean-up (optional)

- Delete the index

```
DELETE myindex
```

Documentation

- Painless Scripting
- Runtime Fields

Example 3: Calculating the age difference in years based on date fields

Requirements

- Create a mapping to the `people` index
- Define a search query that utilizes a runtime field (`current_age`) to calculate the age difference in years between two date fields (`date_of_birth` and `current_date`) within the search results.

Steps

1. **Open the Kibana Console** or use a REST client
2. Create the index

```
PUT people
{
  "mappings": {
    "properties": {
      "date_of_birth": {
        "type": "date"
      },
      "current_date": {
        "type": "date"
      }
    },
    "runtime": {
      "current_age": {
        "type": "long",
        "script": {
          "source": """
          int birthday_year =
 ↪  ZonedDateTime.parse(doc["date_of_birth"].value.toString()).getYear();
          int today_year =
 ↪  ZonedDateTime.parse(doc["current_date"].value.toString()).getYear();
          long age = today_year - birthday_year;
          emit(age);
          """
        }
      }
    }
  }
}
```

3. Index sample documents

```
POST /people/_bulk
{ "index": { "_index": "people", "_id": "1" } }
{ "name": "Alice", "date_of_birth": "1990-01-01", "current_date": "2024-07-08" }
{ "index": { "_index": "people", "_id": "2" } }
{ "name": "Bob", "date_of_birth": "1985-05-15", "current_date": "2024-07-08" }
{ "index": { "_index": "people", "_id": "3" } }
{ "name": "Charlie", "date_of_birth": "2000-12-25", "current_date": "2024-07-08" }
```

4. Construct a search query and return the runtime field:

```
GET people/_search
{
  "query": {
    "match_all": {}
  },
  "fields": [
    "current_age"
  ]
}
```

Test

1. Ensure the documents in your index have `date_of_birth` and `current_date` fields in a compatible date format

```
GET people/_search
```

2. Run the search query and examine the response. The results should include an additional field named `current_age` representing the calculated age difference in years for each document.

```
GET people/_search
{
  "query": {
    "match_all": {}
  },
  "fields": [
```

```
    "current_age"
  ]
}
```

```
// edited responses
{
  ...
    "hits": [
      {
        "_index": "people",
        "_id": "1",
        "_score": 1,
        "_source": {
          "name": "Alice",
          "date_of_birth": "1990-01-01",
          "current_date": "2024-07-08"
        },
        "fields": {
          "current_age": [
            34
          ]
        }
      },
      {
        "_index": "people",
        "_id": "2",
        "_score": 1,
        "_source": {
          "name": "Bob",
          "date_of_birth": "1985-05-15",
          "current_date": "2024-07-08"
        },
        "fields": {
          "current_age": [
            39
          ]
        }
      },
      {
        "_index": "people",
        "_id": "3",
        "_score": 1,
```

```
        "_source": {
          "name": "Charlie",
          "date_of_birth": "2000-12-25",
          "current_date": "2024-07-08"
        },
        "fields": {
          "current_age": [
            24
          ]
        }
      }
    ]
  }
}
```

Considerations

- The runtime field definition utilizes Painless scripting to perform the age calculation.
- The script calculates the difference in years between `current_date` and `date_of_birth` to determine the user's age.

Clean-up (optional)

- Delete the index

```
DELETE people
```

Documentation

- Painless Scripting
- Runtime Fields

5 Cluster Management

5.1 Task: Diagnose shard issues and repair a cluster's health

While the odds are rather high that you will have some unassigned shards if you have done enough of the examples and not cleaned up after yourself we will artifically create some so the below will make some degree of sense.

Example 1: Identifying and resolving unassigned shards to improve cluster health

Requirements

- Identify the cause of unassigned shards.
- Reassign shards to nodes to improve cluster health.
- Ensure all indices are properly allocated and the cluster health status is green.

Steps

1. **Open the Kibana Console** or use a REST client
2. Create an index that needs more replicas than available nodes

```
PUT /a-bad-index
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 2
  }
}
```

3. Check the cluster health and identify unassigned shards (you should see at least 2 unassigned shards)

```
GET /_cluster/health
```

```
{
  "cluster_name": "cluster-1",
  "status": "yellow",
  "timed_out": false,
  "number_of_nodes": 1,
  "number_of_data_nodes": 1,
  "active_primary_shards": 36,
  "active_shards": 36,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 2,
  "delayed_unassigned_shards": 0,
  "number_of_pending_tasks": 0,
  "number_of_in_flight_fetch": 0,
  "task_max_waiting_in_queue_millis": 0,
  "active_shards_percent_as_number": 94.73684210526315
}
```


Figure 5.1: Number of unassigned shards

4. List the unassigned shards (you should see the index name created above with 2 messages of UNASSIGNED)

```
GET
 ↪  _cat/shards?v=true&h=index,shard,prirep,state,node,unassigned.reason&s=state
```


```
index                                                                shard prirep state      node      unassigned.reason
a-bad-index                                                          0     r      UNASSIGNED           INDEX_CREATED
a-bad-index                                                          0     r      UNASSIGNED           INDEX_CREATED
.ds-ilm-history-7-2024.06.25-000002                                  0     p      STARTED    c1-node-1
.slo-observability.summary-v3.2.temp                                 0     p      STARTED    c1-node-1
.ds-.kibana-event-log-ds-2024.06.25-000002                           0     p      STARTED    c1-node-1
.security-7                                                          0     p      STARTED    c1-node-1
.apm-custom-link                                                     0     p      STARTED    c1-node-1
.ds-.kibana-event-log-ds-2024.06.18-000001                           0     p      STARTED    c1-node-1
.kibana-observability-ai-assistant-kb-000001                         0     p      STARTED    c1-node-1
.internal.alerts-ml.anomaly-detection-health.alerts-default-000001   0     p      STARTED    c1-node-1
```


Figure 5.2: Names of the unassigned shards

5. Identify the reason for the unassigned shards

```
GET _cluster/allocation/explain
{
  "index": "a-bad-index",
  "shard": 0,
  "primary": true
}
```

6. In the scenario where you are running an Elasticsearch cluster locally and only have one node then you simply have to lower the number of replicas

```
PUT /a-bad-index/_settings
{
  "index": {
    "number_of_replicas": 0
  }
}
```

Running the shard check again will show that the unassigned shards are now gone.

```
GET
 ↪  _cat/shards?v=true&h=index,shard,prirep,state,node,unassigned.reason&s=state
```

Figure 5.3: The unassigned shards are gone

7. Verify the cluster health again

```
GET /_cluster/health
```

Test

1. Check the cluster health status

```
GET /_cluster/health
```

2. Ensure there are no unassigned shards

```
GET /_cat/shards?v&h=index,shard,prirep,state,unassigned.reason,node
```

Considerations

- The cluster reroute command should be used carefully, especially when accepting data loss.
- Force merging should be done during low traffic periods as it is resource-intensive.
- Regularly monitoring cluster health can prevent shard allocation issues.

Documentation

Start here:

- Diagnose Unassigned Shards
- Red or Yellow Cluster Health Status

For more detail:

- Cat Shards API
- Cluster Health API
- Cluster Reroute API
- Force Merge API
- Nodes Stats API
- Troubleshooting

Example 2: Identifying and resolving a shard failure in a cluster

Requirements

- Identify the cause of a shard failure
- Resolve the issue and restore the cluster's health

Steps

1. **Open the Kibana Console** or use a REST client
2. Check the cluster's health

```
GET /_cluster/health
```

3. Identify the index and shard with issues

```
GET /_cat/shards
```

4. Check the shard's status

```
GET /_cat/shards/{index_name}-{shard_number}
```

5. Resolve the issue (e.g., restart a node, reassign the shard)

```
POST /_cluster/reroute
{
  "commands": [
    {
      "move": {
        "index": "{index_name}",
        "shard": {shard_number},
        "from_node": "{node_name}",
        "to_node": "{new_node_name}"
      }
    }
  ]
}
```

6. Verify the cluster's health

```
GET /_cluster/health
```

Test

- Verify that the shard is no longer in a failed state

```
GET /_cat/shards/{index_name}-{shard_number}
```

Considerations

- Regularly monitoring the cluster's health can help identify issues before they become critical.
- Understanding the cause of the shard failure is crucial to resolving the issue effectively.

Documentation

- Cluster Health
- Cluster-level Shard Allocation and Routing Settings

5.2 Task: Backup and restore a cluster and/or specific indices

This example is specific to a local Elasticsearch cluster, not the Elastic Cloud version. I would recommend learning how to set up a backup and restore from the Kibana UI at **Home > Management > Data > Snapshot and Restore**. I didn't find any documentation on the use of the Kibana dashboard to perform backups and restores.

Example 1: Create a snapshot of multiple indices and and restore them

This example is specific to a local Elasticsearch cluster, not the Elastic Cloud version. I would recommend learning how to set up a backup and restore from the Kibana UI at **Home > Management > Data > Snapshot and Restore**. I didn't find any documentation on the use of the Kibana dashboard to perform backups and restores.

Requirements

- Back up the entire Elasticsearch cluster (all the indices on the cluster)
- Restore specific indices from the backup

Steps

1. *(Do this is you haven't already done it due to a previous exercise)* Configure the **es01** container instance with a backups directory
    1. In a terminal execute bash on the docker container

    ```
    sudo docker exec -it es01 /bin/bash
    ```

    2. Create a backup directory in the current directory of the container

    ```
    mkdir backups
    ```

    If you change directory to `backups` and run `pwd` you'll find that the full path is `/usr/share/elasticsearch/backups`.

    3. Exit the container shell

    ```
    exit
    ```

    4. Update the elasticsearch.yml `path.repo` variable and restart the cluster

1. When we created two single-node clusters (Appendix: Setting Up An Additional Single-node Cluster for Cross-cluster Search (CCS)) we renamed the YAML files for the two cluster:

   - elasticsearch-es01.yml
   - elasticsearch-es02.yml

   For the purposes of this example update **elasticsearch-es01.yml**.

```
path.repo: ["/usr/share/elasticsearch/backups"]
```

2. Copy the YAML file back into the container

```
sudo docker cp elasticsearch-es01.yml
 ↪  es01:/usr/share/elasticsearch/config/elasticsearch.yml
```

3. Restart **es01**

2. **Open the Kibana Console** or use a REST client.
3. Create two sample indexes with some data

```
POST /_bulk
{ "index": { "_index": "example_index1", "_id": "1" } }
{ "name": "Document 1.1" }
{ "index": { "_index": "example_index1", "_id": "2" } }
{ "name": "Document 1.2" }

POST /_bulk
{ "index": { "_index": "example_index2", "_id": "1" } }
{ "name": "Document 2.1" }
{ "index": { "_index": "example_index2", "_id": "2" } }
{ "name": "Document 2.2" }
```

Confirm the documents were indexed

```
GET example_index*/_search
```

4. Create a snapshot repository

```
PUT /_snapshot/example_index_backup
{
    "type": "fs",
    "settings": {
        "location": "/usr/share/elasticsearch/backups"
```

```
    }
}
```

5. Create a snapshot of the two example indices

```
PUT /_snapshot/example_index_backup/snapshot_1
{
    "indices": "example_index1,example_index2",
    "ignore_unavailable": true,
    "include_global_state": false
}
```

6. Verify the snapshot status

```
GET /_snapshot/example_index_backup/snapshot_1
```

7. Delete the two known indices

```
DELETE /example_index1
DELETE /example_index2
```

Check that the two indexes are gone.

```
GET /example_index*/_search
```

8. Restore both indices from the snapshot

```
POST /_snapshot/example_index_backup/snapshot_1/_restore
```

Confirm both indices were restored

```
GET /example_index*/_search
```

9. Restore one index from the snapshot

```
DELETE /example_index1
DELETE /example_index2
```

```
POST /_snapshot/example_index_backup/snapshot_1/_restore
{
  "indices": "example_index2",
  "ignore_unavailable": true,
  "include_global_state": false
}
```

Test

1. Verify the index has been restored

```
GET /example_index2/_search
```

2. Verify the integrity of the snapshot

```
POST /_snapshot/example_index_backup/_verify
```

3. Check the cluster health to ensure the index is properly allocated

```
GET /_cluster/health/example_index2
```

Considerations

- The snapshot repository is configured with the fs (file system) type, which stores the backup data in the container's local file system. For production use, you may want to use a more suitable repository type, such as s3 or gcs.
- The snapshot name snapshot_1 is used to create a backup of the two indices.

Clean-up (optional)

- Delete the indices

```
DELETE /example_index1
DELETE /example_index2
```

- Delete the Backup

```
DELETE /_snapshot/example_index_backup/snapshot_1
```

Documentation

- Snapshot and Restore
- Snapshot Repository APIs
- Snapshot Restore API

Example 2: Create a snapshot of an entire cluster and restore a single index

Requirements

- Back up the entire Elasticsearch cluster
- Restore specific indices from the backup

Steps

1. *(Do this is you haven't already done it due to a previous exercise)* Configure the **es01** container instance with a backups directory

    1. In a terminal execute bash on the docker container

    ```
    sudo docker exec -it es01 /bin/bash
    ```

    2. Create a backup directory in the current directory of the container

    ```
    mkdir backups
    ```

    If you change directory to `backups` and run `pwd` you'll find that the full path is `/usr/share/elasticsearch/backups`.

    3. Exit the container shell

    ```
    exit
    ```

    4. Update the elasticsearch.yml `path.repo` variable and restart the cluster

        1. When we created two single-node clusters (Appendix: Setting Up An Additional Single-node Cluster for Cross-cluster Search (CCS)) we renamed the YAML files for the two cluster:

            - elasticsearch-es01.yml
            - elasticsearch-es02.yml

        For the purposes of this example update **elasticsearch-es01.yml**.

    ```
    path.repo: ["/usr/share/elasticsearch/backups"]
    ```

        2. Copy the YAML file back into the container

    ```
    sudo docker cp elasticsearch-es01.yml
    ↪  es01:/usr/share/elasticsearch/config/elasticsearch.yml
    ```

        3. Restart **es01**

2. **Open the Kibana Console** or use a REST client.
3. Create two sample indexes with some data

```
POST /_bulk
{ "index": { "_index": "example_index1", "_id": "1" } }
{ "name": "Document 1.1" }
{ "index": { "_index": "example_index1", "_id": "2" } }
{ "name": "Document 1.2" }

POST /_bulk
{ "index": { "_index": "example_index2", "_id": "1" } }
{ "name": "Document 2.1" }
{ "index": { "_index": "example_index2", "_id": "2" } }
{ "name": "Document 2.2" }
```

Confirm the documents were indexed

```
GET example_index*/_search
```

4. Create a snapshot repository

```
PUT /_snapshot/example_cluster_backup
{
    "type": "fs",
    "settings": {
        "location": "/usr/share/elasticsearch/backups"
    }
}
```

5. Create a snapshot of the entire cluster

```
PUT /_snapshot/example_cluster_backup/full_cluster_backup
```

6. Verify the snapshot status

```
GET /_snapshot/example_cluster_backup/full_cluster_backup
```

7. Delete one of the existing indices

```
DELETE example_index2
```

8. Restore that specific index from the snapshot with a different name

```
POST /_snapshot/example_cluster_backup/full_cluster_backup/_restore
{
```

```
    "indices": "example_index2",
    "rename_pattern": "example_index2",
    "rename_replacement": "restored_example_index2"
}
```

Test

1. Verify the index has been restored

```
GET /restored_example_index2/_search
```

   The response should include the documents from the original `example_index2`.

2. Optionally, you can delete the original index and verify that the restored index remains

```
DELETE /example_index2
```

```
GET /restored_example_index2/_search
```

3. Verify the integrity of the snapshot

```
POST /_snapshot/example_cluster_backup/_verify
```

4. Check the cluster health to ensure the index is properly allocated

```
GET /_cluster/health/restored_example_index2
```

Considerations

- The snapshot repository is configured with the **fs** (file system) type, which stores the backup data in the container's local file system. For production use, you may want to use a more suitable repository type, such as s3 or gcs.
- The snapshot name full_cluster_backup is used to create a backup of the entire cluster.
- During the restore process, the `rename_pattern` and `rename_replacement` options are used to rename the restored index to `restored_example_index2`.

Clean-up (optional)

- Delete the indices

```
DELETE /example_index1
DELETE /example_index2
DELETE /restored_example_index2
```

- Delete the backup

```
DELETE /_snapshot/example_cluster_backup/full_cluster_backup
```

Documentation

- Snapshot and Restore
- Snapshot Repository APIs
- Snapshot Restore API

Example 3: Creating a snapshot of a single index and restoring it

Requirements

- Create a repository for storing snapshots.
- Take a snapshot of the specified index.
- Restore the snapshot to the cluster.
- Verify the integrity and availability of the restored data.

Steps

1. *(Do this is you haven't already done it due to a previous exercise)* Configure the **es01** container instance with a backups directory

    1. In a terminal execute bash on the docker container

    ```
    sudo docker exec -it es01 /bin/bash
    ```

    2. Create a backup directory in the current directory of the container

    ```
    mkdir backups
    ```

    If you change directory to `backups` and run `pwd` you'll find that the full path is `/usr/share/elasticsearch/backups`.

    3. Exit the container shell

    ```
    exit
    ```

    4. Update the elasticsearch.yml `path.repo` variable and restart the cluster

        1. When we created two single-node clusters (Appendix: Setting Up An Additional Single-node Cluster for Cross-cluster Search (CCS)) we renamed the YAML files for the two cluster:

            - elasticsearch-es01.yml
            - elasticsearch-es02.yml

        For the purposes of this example update **elasticsearch-es01.yml**.

    ```
    path.repo: ["/usr/share/elasticsearch/backups"]
    ```

        2. Copy the YAML file back into the container

    ```
    sudo docker cp elasticsearch-es01.yml
    ↪  es01:/usr/share/elasticsearch/config/elasticsearch.yml
    ```

3. Restart **es01**

2. **Open the Kibana Console** or use a REST client

3. Create two sample indexes with some data

```
POST /_bulk
{ "index": { "_index": "example_index1", "_id": "1" } }
{ "name": "Document 1.1" }
{ "index": { "_index": "example_index1", "_id": "2" } }
{ "name": "Document 1.2" }

POST /_bulk
{ "index": { "_index": "example_index2", "_id": "1" } }
{ "name": "Document 2.1" }
{ "index": { "_index": "example_index2", "_id": "2" } }
{ "name": "Document 2.2" }
```

Confirm the documents were indexed

```
GET example_index*/_search
```

4. Create a snapshot repository

```
PUT /_snapshot/single_index_backup
{
    "type": "fs",
    "settings": {
        "location": "/usr/share/elasticsearch/backups"
    }
}
```

5. Take a snapshot of the specific index

```
PUT /_snapshot/single_index_backup/snapshot_1
{
    "indices": "example_index1",
    "ignore_unavailable": true,
    "include_global_state": false
}
```

6. Verify the snapshot status

```
GET /_snapshot/single_index_backup/snapshot_1
```

7. Delete the index to simulate data loss (optional for testing restore)

```
DELETE /example_index1
```

8. Restore the snapshot

```
POST /_snapshot/single_index_backup/snapshot_1/_restore
{
    "indices": "example_index1",
    "ignore_unavailable": true,
    "include_global_state": false
}
```

Test

1. Verify the index has been restored

```
GET /example_index1/_search
```

2. Verify the integrity of the snapshot

```
POST /_snapshot/single_index_backup/_verify
```

3. Check the cluster health to ensure the index is properly allocated

```
GET /_cluster/health/example_index1
```

Considerations

- The repository type fs is used for simplicity; other types like s3 can be used depending on the environment.
- `ignore_unavailable` ensures the snapshot process continues even if some indices are missing.
- `include_global_state` is set to false to avoid restoring cluster-wide settings unintentionally.

Clean-up (optional)

- Delete the indices

```
DELETE /example_index1
DELETE /example_index2
DELETE /restored_example_index2
```

- Delete the Backup

```
DELETE /_snapshot/single_index_backup/snapshot_1
```

Documentation

- Snapshot and Restore
- Create Snapshot API
- Restore Snapshot API

5.3 Task: Configure a snapshot to be searchable

Example 1: Creating a searchable snapshot for the product catalog index

This can also be done through the **Home > Management > Data > Index Lifecycle Mangement UI**. Again, no documentation on how to perform this using the Kibana dashboard.

Sigh. This will only work if you have an **Enterprise** license.

Requirements

- Create a repository for storing snapshots.
- Take a snapshot of the specified index.
- Mount the snapshot as a searchable index.
- Verify the index is searchable without restoring it to the cluster.

Steps

1. *(Do this is you haven't already done it due to a previous exercise)* Configure the **es01** container instance with a backups directory
    1. In a terminal execute bash on the docker container

    ```
    sudo docker exec -it es01 /bin/bash
    ```

    2. Create a backup directory in the current directory of the container

    ```
    mkdir backups
    ```

    If you change directory to `backups` and run `pwd` you'll find that the full path is `/usr/share/elasticsearch/backups`.

    3. Exit the container shell

    ```
    exit
    ```

    4. Update the elasticsearch.yml `path.repo` variable and restart the cluster
        1. When we created two single-node clusters (Appendix: Setting Up An Additional Single-node Cluster for Cross-cluster Search (CCS)) we renamed the YAML files for the two cluster:
            - elasticsearch-es01.yml
            - elasticsearch-es02.yml

For the purposes of this example update **elasticsearch-es01.yml**.

```
path.repo: ["/usr/share/elasticsearch/backups"]
```

2. Copy the YAML file back into the container

```
sudo docker cp elasticsearch-es01.yml
 ↪  es01:/usr/share/elasticsearch/config/elasticsearch.yml
```

3. Restart **es01**

2. **Open the Kibana Console** or use a REST client.
3. Create a sample index with some data

```
POST _bulk
{ "index": { "_index": "products", "_id": "1" } }
{ "name": "Laptop", "category": "Electronics", "price": 999.99, "stock":
 ↪  50, "description": "A high-performance laptop with 16GB RAM and
 ↪  512GB SSD." }
{ "index": { "_index": "products", "_id": "2" } }
{ "name": "Smartphone", "category": "Electronics", "price": 699.99,
 ↪  "stock": 100, "description": "A latest model smartphone with a
 ↪  stunning display and powerful processor." }
{ "index": { "_index": "products", "_id": "3" } }
{ "name": "Headphones", "category": "Accessories", "price": 199.99,
 ↪  "stock": 200, "description": "Noise-cancelling over-ear headphones
 ↪  with superior sound quality." }
{ "index": { "_index": "products", "_id": "4" } }
{ "name": "Coffee Maker", "category": "Home Appliances", "price": 89.99,
 ↪  "stock": 75, "description": "A programmable coffee maker with a
 ↪  12-cup capacity." }
{ "index": { "_index": "products", "_id": "5" } }
{ "name": "Running Shoes", "category": "Footwear", "price": 129.99,
 ↪  "stock": 150, "description": "Lightweight running shoes with
 ↪  excellent cushioning and support." }
{ "index": { "_index": "products", "_id": "6" } }
{ "name": "Backpack", "category": "Accessories", "price": 49.99,
 ↪  "stock": 300, "description": "Durable backpack with multiple
 ↪  compartments and ergonomic design." }
```

Confirm the documents were indexed

```
GET products/_search
```

4. Create a snapshot repository

```
PUT /_snapshot/products_index_backup
{
    "type": "fs",
    "settings": {
        "location": "/usr/share/elasticsearch/backups"
    }
}
```

5. Take a snapshot of the specific index

```
PUT /_snapshot/products_index_backup/snapshot_1
{
  "indices": "products",
  "ignore_unavailable": true,
  "include_global_state": false
}
```

6. Verify the snapshot status

```
GET /_snapshot/products_index_backup/snapshot_1
```

7. Delete the index to simulate data loss (optional for testing restore)

```
DELETE /products
```

8. Mount the snapshot as a searchable index

```
PUT /_snapshot/products_index_backup/snapshot_1/_mount
{
    "index": "products",
    "renamed_index": "products_backup_searchable"
}
```

If you don't have an **Enterprise** license the above will fail.

Test

1. Verify the mounted index is searchable

```
GET /products_backup_searchable/_search
```

2. Check the cluster health to ensure the searchable snapshot is properly allocated

```
GET /_cluster/health/products_backup_searchable
```

Considerations

- The repository type fs is used for simplicity; other types like s3 can be used depending on the environment.
- `ignore_unavailable` ensures the snapshot process continues even if some indices are missing.
- `include_global_state` is set to false to avoid restoring cluster-wide settings unintentionally.
- Mounting the snapshot as a searchable index allows for searching the data without the need to fully restore it, saving resources and time.

Clean-up (optional)

- Delete the index

```
DELETE /products
```

- Delete the Backup

```
DELETE /_snapshot/products_index_backup/snapshot_1
```

Documentation

- Create Snapshot API
- Mount Searchable Snapshot API
- Search API
- Searchable Snapshot
- Snapshot and Restore

5.4 Task: Configure a cluster for cross-cluster search

FYI: This is similar to the example at *Searching Data > Write and execute a query that searches across multiple clusters*

Example 1: Setting up cross-cluster search between a local cluster and a remote cluster for an e-commerce catalog

The following instructions are for two single-node clusters running locally on your computer.

Requirements

- Configure the remote cluster to be searchable from the local cluster.
- Ensure secure communication between clusters.
- Verify the cross-cluster search functionality.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Configure the remote cluster on the local cluster

```
PUT /_cluster/settings
{
"persistent": {
    "cluster": {
    "remote": {
        "es01": {
        "seeds": [
            "es01:9300"
        ],
        "skip_unavailable": true
        },
        "es02": {
        "seeds": [
            "es02:9300"
        ],
        "skip_unavailable": false
        }
    }
    }
```

```
}
}
```

3. (optional if you are doing this locally) Set up security settings where you have keystores properly setup. On the remote cluster:

```
PUT /_cluster/settings
{
    "persistent": {
        "xpack.security.enabled": true,
        "xpack.security.transport.ssl.enabled": true,
        "xpack.security.transport.ssl.verification_mode": "certificate",
        "xpack.security.transport.ssl.keystore.path":
        ↪ "/path/to/keystore.jks",
        "xpack.security.transport.ssl.truststore.path":
        ↪ "/path/to/truststore.jks"
    }
}
```

On the local cluster:

```
PUT /_cluster/settings
{
    "persistent": {
        "xpack.security.enabled": true,
        "xpack.security.transport.ssl.enabled": true,
        "xpack.security.transport.ssl.verification_mode": "certificate",
        "xpack.security.transport.ssl.keystore.path":
        ↪ "/path/to/keystore.jks",
        "xpack.security.transport.ssl.truststore.path":
        ↪ "/path/to/truststore.jks"
    }
}
```

4. Verify the remote cluster configuration

```
GET /_remote/info
```

5. Index product documents into each cluster.

- For **es01** (potentially the local cluster):

```
POST /products/_bulk
{ "index": { "_id": "1" } }
```

```
{ "product": "Elasticsearch Guide", "category": "Books", "price": 29.99 }
{ "index": { "_id": "2" } }
{ "product": "Advanced Elasticsearch", "category": "Books", "price": 39.99 }
{ "index": { "_id": "3" } }
{ "product": "Elasticsearch T-shirt", "category": "Apparel", "price": 19.99 }
{ "index": { "_id": "4" } }
{ "product": "Elasticsearch Mug", "category": "Apparel", "price": 12.99 }
```

- For **es02** (potentially the "remote" cluster) through the command line:

```
curl -u elastic:[your password here] -X POST "http://localhost:9201/products/_bulk?pretty" -H 'Content-Type: application/json' -d'
{ "index": { "_id": "5" } }
{ "product": "Elasticsearch Stickers", "category": "Accessories", "price": 4.99 }
{ "index": { "_id": "6" } }
{ "product": "Elasticsearch Notebook", "category": "Stationery", "price": 7.99 }
{ "index": { "_id": "7" } }
{ "product": "Elasticsearch Pen", "category": "Stationery", "price": 3.49 }
{ "index": { "_id": "8" } }
{ "product": "Elasticsearch Hoodie", "category": "Apparel", "price": 45.99 }    '
```

6. Perform a cross-cluster search query

```
GET /remote_cluster:products/_search
```

Test

1. Verify the remote cluster info

```
GET /_remote/info
```

2. Search the remote cluster from the local cluster

```
GET /remote_cluster:product_catalog/_search
```

Considerations

- Ensure that the nodes listed in the seeds setting are accessible from the local cluster.
- Security settings such as SSL/TLS should be configured to ensure secure communication between clusters.
- Regularly monitor the connection status between the clusters to ensure reliability.

Clean-up (optional)

- Delete the **es01** index.

```
DELETE products
```

- Delete the **es02** index from the command line.

```
curl -u elastic:[your password here] -X DELETE
 ↪  "http://localhost:9201/products?pretty"
```

Documentation

- Cross-Cluster Search
- Cluster Remote Info API
- Search API
- Security Settings

5.5 Task: Implement cross-cluster replication

There are a number of ways to set up cross-cluster replication and they can all be found here.

Cross-cluster replication needs an **Enterprise** license

Example 1: Setting up cross-cluster replication for the product catalog index between a leader cluster and a follower cluster

In this example, we will run 2 single-node clusters locally using containers (as we have for all the other examples).

- The **es01** container instance will be considered
  - leader
  - remote
- The **es02** container instance will be considered
  - follower
  - local

You may also need to get a **free 30-day trial license** of certain features including cross-cluster replication. Since the second cluster is not hooked up to Kibana execute this from the command line (assuming you called the docker instance **es02** as we have been using in this guide):

```
curl -v -u elastic:[YOUR ELASTIC PASSWORD HERE] -X POST
 ↪  "http://localhost:9201/_license/start_trial?pretty&acknowledge=true"
```

Requirements

- Configure remote cluster settings on both leader and follower clusters.
- Set up the leader index on the leader cluster.
- Configure the follower index on the follower cluster to replicate from the leader index.
- Ensure secure communication between clusters.
- Verify replication and data consistency.

Steps

1. **Open the Kibana Console** or use a REST client.
2. Configure the remote cluster settings on the **leader** cluster (**es01**)

```
PUT /_cluster/settings
{
  "persistent": {
    "cluster": {
      "remote": {
        "es01": {
        "seeds": [
            "es01:9300"
        ],
        "skip_unavailable": true
        },
        "es02": {
        "seeds": [
            "es02:9300"
        ],
        "skip_unavailable": false
        }
      }
    }
  }
}
```

3. Configure the local cluster settings on the **follower** cluster (**es02**)

```
curl -v -u elastic:[YOUR ELASTIC PASSWORD HERE] -X PUT
 ↪  "http://localhost:9201/_cluster/settings?pretty" -H "Content-Type:
 ↪  application/json" -d'
{
  "persistent": {
    "cluster": {
      "remote": {
        "es01": {
        "seeds": [
            "es01:9300"
        ],
        "skip_unavailable": true
        },
        "es02": {
```

```
        "seeds": [
            "es02:9300"
        ],
        "skip_unavailable": false
        }
      }
    }
  }
}'
```

4. Create the leader index on the **leader** cluster (**es01**)

```
PUT /product_catalog
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
     "properties": {
     "product_id": {
       "type": "keyword"
     },
     "name": {
       "type": "text"
     },
     "description": {
       "type": "text"
      },
      "price": {
        "type": "double"
      }
    }
  }
}
```

5. Index sample documents in the **leader** index

```
POST /product_catalog/_bulk
{ "index": { "_id": "1" } }
{ "product_id": "p001", "name": "Product 1", "description": "Description
 ↪  of product 1", "price": 20.0 }
{ "index": { "_id": "2" } }
```

```
{ "product_id": "p002", "name": "Product 2", "description": "Description
 ↪  of product 2", "price": 30.0 }
```

6. Configure the follower index on the **follower** cluster through the command line

```
curl -v -u elastic:[YOUR ELASTIC PASSWORD HERE] -X PUT
 ↪  "http://localhost:9201/product_catalog_follower/_ccr/follow?pretty"
 ↪  -H "Content-Type: application/json" -d'
{
  "remote_cluster": "es01",
  "leader_index": "product_catalog"
}'
```

Test

1. Verify the **follower** index (**es02**) is following the leader index

```
curl -v -u elastic:[YOUR ELASTIC PASSWORD HERE]
 ↪  "http://localhost:9201/product_catalog_follower/_stats?pretty"
```

2. Check the data in the **follower** index (**es02**) to ensure it matches the **leader** (**es01**) index

```
curl -v -u elastic:[YOUR ELASTIC PASSWORD HERE]
 ↪  "http://localhost:9201/product_catalog_follower/_search?pretty"
```

Considerations

- Ensure the nodes listed in the seeds setting are accessible from the follower cluster.
- Security settings such as SSL/TLS should be configured to ensure secure communication between clusters (but not for this example given the YAML changes suggested in the *Appendix*).
- Regularly monitor the replication status and performance to ensure data consistency and reliability.

Clean-up (optional)

- Delete the **follower** configuration

```
curl -v -u elastic:[YOUR ELASTIC PASSWORD HERE] -X DELETE
 ↪  "http://localhost:9201/product_catalog_follower?pretty"
```

- Delete the index

```
DELETE product_catalog
```

Documentation

- Cross-Cluster Replication
- Create Follower Index API
- Cluster Remote Info API
- Search API
- Security Settings