JOHN BRYCE
Leading in IT Education
matrix company

UPDATED TO ELK V.8.3.3

PART 3 SEARCHING WITH ES

●●●

Provided by DevOpShift & Autherd by Yaniv Cohen
DevopShift CEO and Head of Devops JB

- INTRODUCTION -

I

Introduction

Elasticsearch, beyond its powerful search capabilities, offers a comprehensive API that enables users to interact, manage, and maintain their clusters. This API provides invaluable insights, utilities, and tools for seamless Elasticsearch operations, ensuring optimal performance, scalability, and reliability.

- TOPICS COVERED IN PART 3 -

Cluster Health & Status Basic Overview:

- Dive into API endpoints that offer a glimpse into the cluster's health, ensuring its robust operation.

Indexing Data in ES:

- Powering Search: From Data Ingestion to Index Creation

Basic Search Queries in ES:

- Understand the intricacies of crafting effective search queries to retrieve meaningful data.

Reindexing process and scenarios:

- Powering Search: From Data Ingestion to Index Creation

Schema Management:

- Flexibility and Structure: Managing Dynamic Data Models

As we delve into Part 3, we'll uncover the internals of Elasticsearch's API, equipping you with the knowledge to harness its full potential, whether it's extracting insights, ensuring system health, or pre-emptively addressing potential issues.

- Cluster Health & Status - BASIC OVERVIEW

Deep dive for the following part will be discussed in our Operation part if included in your learning path

H

HEALTH & STATUS

Elasticsearch's cluster health status provides a quick snapshot of the overall well-being of your cluster. It is a primary metric to determine if your data is safe, if your cluster is fully operational, or if any immediate action is required.

- Understanding Health Colors -

Cluster status colors

Green Status:

- **Meaning**: All primary and replica shards are active.
- **Implication**: Your cluster is fully operational, and data is secure.

Yellow Status:

- **Meaning**: All primary shards are active, but not all replica shards are allocated.
- **Implication**: Data is still intact and searchable, but you're at a higher risk.
  If a node fails, data might be lost.

Red Status:

- **Meaning**: One or more primary shards are not active. This could mean data is not available for these shards.
- **Implication**: Immediate action is required. Some data is currently inaccessible.

- Factors Influencing Cluster Health -

H

Overview - Cluster status degradation reasons

Shard Allocation:
- The process of allocating shard replicas to nodes. Issues here can lead to yellow or red statuses.

Node Availability:
- If nodes containing primary shards are down, the cluster health will degrade.

Cluster Settings & Constraints:
- Certain settings might prevent shard allocation, influencing cluster health.

External Factors:
- Issues like network partitions or hardware failures can have direct impacts on health.

Importance of Monitoring Cluster Health

Proactive Issue Detection: Regularly checking cluster health can help identify issues before they escalate.
Data Integrity: Ensures that your data is safe and always available.
Performance Maintenance: A healthy cluster typically performs better, offering faster search and indexing speeds.

In essence, the cluster health status serves as a frontline metric in Elasticsearch operations. It's the first thing admins look at when gauging cluster performance and stability. A clear understanding of what each status means and the factors influencing it is paramount to effective Elasticsearch management.

- Understanding Key Metrics in Cluster Health -

DIVING IN

INTRODUCTION:

In Elasticsearch, a multitude of metrics provide insights into the cluster's operational state. Understanding these metrics is essential to effectively manage and troubleshoot your cluster.

H

Number of Nodes & Data Nodes

Definition:

Nodes: Individual server instances forming the Elasticsearch cluster.
Data Nodes: Nodes that store data and participate in indexing and search operations.

Example

```
{
  "number_of_nodes": 5,
  "number_of_data_nodes": 4
}
```

Implication: Out of 5 nodes, 4 are data nodes. A non-data node might be a dedicated master or ingest node.

Active Shards & Unassigned Shards

Definition:

Active Shards: Shards that are operational, both primary and replicas.
Unassigned Shards: Shards not allocated to any node, often indicating issues.

Example

```
{
  "active_primary_shards": 10,
  "active_shards": 20,
  "unassigned_shards": 2
}
```

Implication: There are 10 primary and 10 replica shards active. 2 shards, possibly replicas, are not allocated.

H

Task Queue Lengths & Pending Tasks

Definition:

Task Queues: Elasticsearch nodes manage tasks like indexing in queues.
Pending Tasks: Operations waiting for execution, indicative of cluster overload or issues.

Example

```
{
  "number_of_in_flight_fetch": 0,
  "number_of_pending_tasks": 3
}
```

Implication: There are no ongoing shard fetches, but 3 tasks are pending execution, signaling potential node overload.

H

Relocating Shards

Definition: Shards being moved from one node to another, often due to node imbalance or failures.

Example

```
{
  "relocating_shards": 2
}
```

Implication: 2 shards are currently being moved. This might be due to a node nearing its disk watermark or cluster rebalancing.

- Delving Deeper using API Endpoints -

Delving Deeper with API Endpoints - overview

```
GET /_cluster/health
```

This endpoint returns a comprehensive view of your cluster's state, providing invaluable data for diagnostics and monitoring.

In conclusion, these key metrics offer a granular view into the cluster's operations. Regularly monitoring and understanding these figures ensures proactive management, helping preempt issues and ensuring smooth cluster operations.

- Indices Health -

INTRODUCTION:

Every index in Elasticsearch has its health status, which can be pivotal for diagnosing issues in larger clusters. Let's delve into the specifics.

- Individual Index Statuses -

H

Individual Index Statuses

Definition: Each index will have a health status (green, yellow, or red), similar to the overall cluster health.

Example:
Using the following API call, you can fetch the health status of individual indices:

```
GET /_cat/indices?v
```

Output might look like:

```
Health  status  index      uuid     pri rep docs.count docs.deleted store.size pri.store.size
Yellow    open  my-index   abc123    5   1     1000          50          20mb       20mb
```

Implication: The my-index is yellow, meaning not all replicas are allocated.

Dealing with Yellow or Red Indices

- **Identify the Cause:** Use the _cluster/allocation/explain API to understand why shards are unassigned.

- **Replica Allocation**: For yellow indices, it might be due to insufficient nodes for replicas.
- **Node Failures**: Red indices can result from node failures, especially if replicas weren't configured.
- **Resolutions**:
    - Add nodes to the cluster.
    - Adjust the number of replicas.
    - Investigate and rectify node failures.

- Components Affecting Cluster Health -

H

INTRODUCTION:

Elasticsearch cluster health can be influenced by a myriad of internal and external components. Understanding these is crucial to proactive management and effective troubleshooting.

- Impact of Node Failures -

Impact of Node Failures

Definition: A node failure occurs when a node in the Elasticsearch cluster becomes unresponsive or crashes.

Consequences:

- Unassigned shards leading to yellow or red cluster health.
- Potential data loss if no replicas were set up.
- Decreased search and indexing capacity.

Resolution:

- Restart the failed node or troubleshoot the root cause.
- Ensure that the cluster has adequate replicas to handle node failures.

H

Impact of Node Failures - API Insight

Definition: A node failure occurs when a node in the Elasticsearch cluster becomes unresponsive or crashes.

Example:

```
GET /_cat/nodes?v
```

Example Output:

```
ip          heap.percent ram.percent cpu     load_1m   node.role    master   name
127.0.0.1       62           55       3      1.75          mdi        *      node-1
127.0.0.2       57           52       4      1.60          mdi        -      node-2
```

Missing nodes from this list can indicate a node failure.

H

Disk Watermarks and Their Influence

Definition: Disk watermarks are thresholds that trigger cluster actions when disk usage rises above certain levels.

Types:

- **Low:** Might prevent the allocation of new shards to the node.
- **High**: Will prevent the allocation of new shards and might start relocating existing ones.
- **Flood Stage**: Will block all writes to indices with shards on the node.

Resolution:

Clear old indices or increase storage.
Adjust watermark settings judiciously.

H

Disk Watermarks - API Insight

Definition: Disk watermarks are thresholds that trigger cluster actions when disk usage rises above certain levels.

Example:

```
GET /_cat/allocation?v
```

Example Output:

```
shards   disk.indices disk.used disk.avail disk.total  disk.percent  host     ip        node
  45       97gb         108gb     241gb      350gb        30         Node-1 127.0.0.1 node-1
  47       99gb         110gb     239gb      350gb        31         Node-2 127.0.0.2 node-2
```

Nodes nearing the disk's capacity may hit watermark thresholds.

H

Network Partitions

Definition: Occurs when nodes in a cluster lose communication with each other.

Consequences:

- "Split-brain" scenarios where multiple nodes might think they are the primary for the same shard.
- Delayed or failed operations.

Resolution:

Use a quorum-based approach for master node elections.
Monitor and ensure network reliability.

H

Network Partitions - API Insight

Definition: Occurs when nodes in a cluster lose communication with each other.

Example:

```
GET /_cat/nodes?v&h=ip,role,master,name
```

Example Output:

```
ip          role  master  name
127.0.0.1   mdi     *     node-1
127.0.0.2   mdi     m     node-2
```

Nodes with conflicting master roles (multiple stars) might indicate network partition issues.

External Factors

Hardware Failures:

- Disk crashes, memory issues, or CPU overloads can lead to node failures.

Resource Starvation:

- Inadequate CPU, memory, or I/O can degrade performance.

Misconfigurations:

- Incorrect node or index settings can lead to various issues.

Overloaded Cluster:

- High query loads or rapid data ingestion can strain the cluster.

Hardware Failures API Call

Hardware Failures:
API Call: Monitoring tools or system metrics would be more suitable here.

Resource Starvation:
API Call:

```
GET /_nodes/stats?filter_path=nodes.*.os
```

Example Output:

```
{
 "nodes": {
  "1kQIR7wSTma2dRM3": {
   "os": {
    "cpu": {"percent": 80},
    "mem": {"used_percent": 90}
   }
```

High CPU or memory usage can indicate resource starvation.

H

Misconfigurations & Overloaded Cluster:

API Call:

```
GET /_cluster/settings
```

Examining cluster settings can provide insights into any potential misconfigurations.

Summary

- These API calls and their respective outputs give an immediate understanding of specific cluster components. Monitoring these metrics helps in the early detection of potential issues and fosters timely resolution.

- API Endpoints for Health & Status -

Summary

- Elasticsearch provides a suite of API endpoints that grant insights into various aspects of the cluster, helping admins monitor, diagnose, and manage it effectively.

H

The Cluster Health API

Definition: An API that provides a high-level overview of the cluster's health status.

API Call:

```
GET /_cluster/health
```

Implication: Cluster is in a yellow state due to unassigned replica shards.

```
{
  "cluster_name": "my-cluster",
  "status": "yellow",
  "number_of_nodes": 2,
  "number_of_data_nodes": 1,
  "active_primary_shards": 5,
  "active_shards": 5,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 5
}
```

H

Cluster Stats API

Definition: Provides detailed statistics about the cluster, including node information, index stats, and more.

API Call:

```
GET /_cluster/stats
```

```
{
  "cluster_name": "my-cluster",
  "status": "yellow",
  "nodes": {
    "count": {"total": 2, "data": 1},
    "versions": ["7.10.2"]
  },
  "indices": {
    "count": 10,
    "shards": {"total": 20, "primaries": 10,
"replication": 1}
  }
}
```

Implication: This output provides a detailed overview of nodes, their versions, and index statistics.

H

Cat APIs for Human-readable Insights

Definition: The cat APIs provide a concise and human-readable way to retrieve essential cluster and index information.

API Call:

```
GET /_cluster/stats
```

Example Output

```
ip          heap.percent ram.percent cpu load_1m node.role master name
127.0.0.1             62          55   3    1.75 mdi          *      node-1
127.0.0.2             57          52   4    1.60 mdi          -      node-2
```

Summary

- In essence, these API endpoints serve as the pulse-check tools for your Elasticsearch cluster. By regularly querying them, one can keep tabs on the cluster's state, ensuring optimal performance and timely interventions when needed.

- Indexing Data in Elasticsearch -

I

INTRODUCTION

Transforming MovieLens Data into Searchable Insights

Elasticsearch's power shines when it comes to indexing and searching vast datasets. In this section, we'll be walking through the process of ingesting the ***MovieLens*** dataset into Elasticsearch, from single movie entries to bulk imports.

- Why Indexing -

I

WHY INDEXING

The Foundation of Search

Search Engine Basics:

- At its core, Elasticsearch is a search engine. It needs to transform raw data into a format

optimized for rapid search and retrieval.

- Indexing: The process by which Elasticsearch converts raw data (like the MovieLens dataset) into this searchable format.

What is an Index - Recap

The Foundation of Search

Definition:

- An index in Elasticsearch is somewhat similar to a table in a database. It has a mapping which defines the types of fields.

Purpose:

- Stores data and allows for fast search operations.

Components:

- Composed of shards, which are the fundamental units of storage and search in Elasticsearch.

The Indexing Process

The Foundation of Search

Data Preparation:

- Transform data into a JSON structure.

Sending Data:

- Use RESTful operations to send data to Elasticsearch.

Internal Operations:

- Elasticsearch tokenizes the data, builds an inverted index, and stores it.

- Basic Index API -

Basic Index API

Endpoint: PUT /<index-name>/_doc/<document-id>

Payload: The JSON representation of your data.
Response: Acknowledgment from Elasticsearch with metadata about the indexed document.

Example:

```
PUT /movies/_doc/1
{
  "title": "Toy Story",
  "genres": ["Animation", "Comedy", "Family"]
}
```

Now let’s understand the the Indexing endpoint in greater details.

Basic Index API - Bulk Indexing

Why Bulk?: For efficiency. Indexing documents one-by-one can be slow, especially for large datasets.

Endpoint: POST /_bulk
Format: Alternating lines of metadata and actual data.

Example Bulk Format:

```
{ "index": { "_index": "movies", "_id": 1 } }
{ "title": "Toy Story", "genres": ["Animation", "Comedy", "Family"] }
{ "index": { "_index": "movies", "_id": 2 } }
{ "title": "Jumanji", "genres": ["Adventure", "Fantasy"] }
```

Basic Index API - Modifying Data: Updates & Deletes

Update API: Allows for modifications to existing documents.

Endpoint: POST /<index-name>/_doc/<document-id>/_update

- Partial updates possible.

Delete API: Removes documents from an index.

- **Endpoint**: DELETE /<index-name>/_doc/<document-id>

Basic Index API - Concurrency Concerns

Optimistic Concurrency Control: Each document has a version. If two updates occur concurrently, only one will succeed.

Use Case: Imagine two users trying to update the rating of a movie at the same time

Now let’s understand the the Indexing endpoint in greater details.

- Index API Understanding the Endpoint -

INDEX API ENDPOINT

The Structure of Elasticsearch's Indexing API

Endpoint Anatomy: /<index-name>/_doc/<document-id>

- **Index Name**: Specifies which index the document should be added to. Just Like a table in a relational database.
- **Document ID:** A unique identifier for each document within the index.

Document ID: Manual vs. Automatic

- **Manual**: You can specify an ID when indexing the document. Useful when your dataset already has unique identifiers (like a movie ID).
- **Automatic**: If you don't specify an ID, Elasticsearch will generate one for you. This is a random UUID, ensuring uniqueness.

INDEX API ENDPOINT

The Structure of Elasticsearch's Indexing API

Considerations:

- **Manual IDs**: Can be more readable and can match external systems. However, there's a risk of collisions if not careful.
- **Automatic IDs**: No risk of collisions, but can be harder to interpret without additional context.

I

INDEX API ENDPOINT

Beyond Data: The Structural Components of an Index

Shards:

- **Definition**: A shard is a self-contained index. When data grows, it can be distributed across multiple shards.

- **Primary Shards**: The original shards where data gets indexed.

- **Why Shards?**: They enable horizontal scaling. Data can be distributed across multiple nodes, allowing Elasticsearch to handle vast datasets.

- **Configuration**: The number of primary shards is set when the index is created and can't be changed later.

I

INDEX API ENDPOINT

Beyond Data: The Structural Components of an Index

Replicas:

- **Definition**: Replicas are copies of primary shards, ensuring data redundancy.
- **Purpose**: They serve two main functions:
- **High Availability**: In case a node or shard fails, replicas ensure that no data is lost.
- **Load Balancing**: Search queries can be handled by primary and replica shards, distributing the load.
- **Configuration**: The number of replicas can be changed dynamically as per the cluster's needs.

I

INDEX API ENDPOINT

Beyond Data: The Structural Components of an Index

Index Settings:

- **Dynamic Settings**: Some settings, like the number of replicas, can be changed after the index is created.

- **Static Settings**: Some settings, like the number of primary shards, are fixed once the index is created.

- **Tuning for Use Case**: Depending on the expected query and indexing load, the settings can be fine-tuned. For instance, an index expected to receive a high search load might benefit from more replicas.

I

INDEX API ENDPOINT

Beyond Data: The Structural Components of an Index

EXAMPLE:
This creates an index "movielens" with 3 primary shards and 2 replica copies for each primary shard.

```
PUT /movielens
{
 "settings": {
  "number_of_shards": 3,
  "number_of_replicas": 2
 }
}
```

- INDEX _SETTING ENDPOINT -

INDEX _SETTING API ENDPOINT

Peeking into Index Configurations

Purpose:

- The _settings API provides insights into the current settings and configurations of an index.

Usage:

- It's often utilized after creating or modifying an index to verify settings or before making modifications to ensure correct configurations.

INDEX _SETTING API ENDPOINT

How to Use the _settings API

Fetch Settings for a Specific Index:

```
GET /<index-name>/_settings
```

This will return the settings for the specified index.

Fetch Settings for All Indexes:

```
GET /_all/_settings
```

This provides a broader view of settings across all indexes

INDEX _SETTING API ENDPOINT

Sample Output & Interpretation

Interpreting the Output:

Number_of_shards:
- Indicates the number of primary shards.

number_of_replicas:
- Indicates the number of replica shards.

creation_date:
- Timestamp of when the index was created.

uuid:
- A unique identifier for the index.

```
{
 "movielens": {
   "settings": {
     "index": {
       "creation_date": "1633450792289",
       "number_of_shards": "3",
       "number_of_replicas": "2",
       "uuid": "I8Zx-z-4T6yslqAhZfRzWA",
       "version": { "created": "7080099" },
       "provided_name": "movielens"
     }
   }
 }
}
```

I

INDEX _SETTING API ENDPOINT

Importance of Monitoring Index Settings

Optimization:

- By regularly monitoring and tweaking settings, performance can be optimized.

Troubleshooting:

- If facing issues, checking settings can often provide insights.

Planning:

- Before scaling or adding more data, understanding current settings can guide decisions.

- Best Practices for Indexing in Elasticsearch -

INDEX API ENDPOINT

Best Practices

Considerations:

- **Predefine Mappings**: Especially when you know the structure of your data (like MovieLens). This ensures data is indexed correctly.

- **Use Bulk Indexing (TBD)**: Particularly for large datasets. This reduces overhead and speeds up the indexing process.
-
- **Monitor Indexing Performance**: Keep an eye on Elasticsearch's performance metrics during indexing. Adjust your indexing rate if needed.
-
- **Choose Document IDs Wisely**: As discussed, consider whether manual or automatic IDs are best for your use case

- Basic Index Structure of Mappings -

I

What Are Mappings:

Laying the Blueprint for Your Data

Definition:

- Mappings define how documents and their fields are stored and indexed in Elasticsearch.

Analogy:

- Think of mappings as the schema in a relational database.

Purpose:

- To inform Elasticsearch about the nature of the data (text, number, date, etc.) and how it should be treated.

I

What Are Mappings:

Structure of Mappings

Field Data Types:

- Such as text, keyword, date, long, double, boolean, etc.

Field Properties:

- You can specify how fields should be analyzed, if they should be searchable, index patterns, etc.

Nested Fields:

- For more complex structures, like lists or arrays.

What Are Mappings:

Basic Example

```
PUT /movielens_with_mapping
{
 "mappings": {
  "properties": {
   "title": { "type": "text" },
   "genres": { "type": "keyword" },
   "release_year": { "type": "integer" }
  }
 }
}
```

- INDEX _MAPING ENDPOINT -

INDEX _MAPPING API ENDPOINT

Understanding Data Structures within an Index

Definition:

- Mappings in Elasticsearch define how documents and their fields are stored and indexed.

Purpose:

- The _mapping API allows users to retrieve these mappings, understanding the structure and types associated with each field in an index.

I

INDEX _MAPPING API ENDPOINT

Utilizing the _mapping API

Fetch Mapping for a Specific Index:

```
GET /<index-name>/_mapping
```

This returns the mapping of the specified index.

Fetch Mapping for All Indexes:

```
GET /_all/_mapping
```

Useful for obtaining a broader perspective on mappings across all indexes.

INDEX _MAPPING API ENDPOINT

Sample Output & Deciphering

Interpreting the Output:
Properties:

- This section lists all the fields within the index.
- Each field (like title, genres) has an associated type which indicates how the data is stored and indexed.

Some types:

- like date, might have additional settings (like format)

```
{
 "movielens": {
  "mappings": {
   "properties": {
    "title": { "type": "text" },
    "genres": { "type": "keyword" },
    "release_date": { "type": "date", "format": "year" }
   }
  }
 }
}
```

I

INDEX _MAPPING API ENDPOINT

Why Monitor Mappings?

Data Integrity:

- Ensures that data being indexed aligns with the expected structures.

Query Planning:

- Understanding mappings is vital for crafting effective search queries.

Modifications:

- Before making changes to an index, understanding the current mappings is essential.

INDEX _MAPPING API ENDPOINT

Changing Mappings

Static vs. Dynamic:

- While many aspects of mappings are static (can't be changed after data is indexed), some parts, like adding new fields, can be dynamic.

Reindexing: (TBD)

- If major changes to mappings are needed, often the solution is to create a new index with the desired mappings and reindex the data.

- Introduction to Reindexing -

Introduction to Reindexing

Reshaping Data within Elasticsearch

Definition:

- Reindexing is the process of copying documents from one index to another. This can involve changing the data structure, modifying content, or simply copying.

Purpose:

- Allows data structure modifications which are otherwise static once indexed, such as changing mappings or settings.

Introduction to Reindexing

Why Reindex?

Changing Mappings:

- If you need to change the structure of the data, reindexing is required because mappings are immutable.

Data Transformation:

- While moving data, you can transform its content using scripts.

Index Upgrades:

- For major version upgrades of Elasticsearch, reindexing is sometimes needed.

Data Pruning:

- Reindexing can be used to filter out unnecessary data and reduce index size.

How to Reindex?

Elasticsearch offers a built-in API for reindexing

Considerations:

Reindexing can be resource-intensive.
It's vital to monitor cluster health during the operation.
Ensure enough disk space is available.

```
POST /_reindex
{
  "source": { "index": "old_index" },
  "dest": { "index": "new_index" }
}
```

Note: This is the basic of the _reindex process.
We will dive into it on the next part

How to Reindex?

Reindexing Challenges & Best Practices

Performance:

- Reindexing is IO and CPU-intensive. It's best done during off-peak times.

Consistency:

- Ensure data isn't being actively modified during the process to maintain consistency.

Backups:

- Always have backups before major reindexing operations.

Validation:

- Post-reindexing, thoroughly validate the data and its structure.

- Introduction to Search in Elasticsearch -

Note:
Deep dive into search will be discussed in the next parts

Introduction to Search in Elasticsearch

Retrieving Data Made Simple and Efficient

Definition:

- Search is the process of querying your data to retrieve relevant documents.

Purpose:

- Allows users to retrieve, analyze, and visualize the data stored within indices.

S

Introduction to Search in Elasticsearch

Harnessing the Power of Elasticsearch

Using the Query String:

```
GET /movielens/_search?q=title:Toy Story
```

This performs a search on the movielens index for documents where the title field contains "Toy Story".

Using the Request Body:

Elasticsearch provides a more powerful and flexible way to define searches using JSON in the request body.

```
GET /movielens/_search
{
  "query": {
    "match": { "title": "Toy Story" }
  }
}
```

Introduction to Search in Elasticsearch

Understanding the Search Response

A sample output might look like:

Key Elements:

took: How many milliseconds the search took.
hits: Contains the search results.

```
{
 "took": 30,
 "timed_out": false,
 "_shards": {
  "total": 5,
  "successful": 5,
  "skipped": 0,
  "failed": 0
 },
 "hits": {
  "total": 1,
  "max_score": 1.3862944,
  "hits": [
    { "_index": "movielens", "_type": "_doc", "_id": "1", "_score":
1.3862944, "_source": { "title": "Toy Story", "genres":
["Animation", "Comedy"] } }
  ]
 }
}
```

- Deeper Dive: Search with JSON Body -

Deeper Dive: Search with JSON Body

Flexibility:
- The JSON body allows for a wide range of queries, from simple to complex.

Combining Queries:
- You can combine multiple queries, apply filters, or even utilize boolean logic.

Advanced Features:
- Aggregations, highlighting, and suggestions can also be added.

```
GET /movielens/_search
{
  "query": {
    "bool": {
      "must": { "match": { "title": "Story" } },
      "filter": { "term": { "genres": "Animation" } }
    }
  }
}
```

This search looks for movies with "Story" in the title that are also in the "Animation" genre.

Basic Search Queries

Match All Query:

- Fetches all documents from an index.

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

Basic Search Queries

Match Query:

- Fetches documents that match specified criteria.

```
GET /movielens/_search
{
 "query": {
  "match": { "title": "Toy Story" }
 }
}
```

S

Basic Search Queries

Search with JSON Request Body

- **Using JSON** in the request body allows for complex and precise query construction.
- **Offers flexibility** with features like filtering, sorting, and aggregations.

```
GET /movielens/_search
{
 "query": {
   "bool": {
     "must": [
       { "match": { "genres": "Animation" } }
     ],
     "filter": [
       { "range": { "release_year": { "gte": "1990" } } }
     ]
   }
 },
 "sort": [
   { "release_year": "desc" }
 ]
}
```

Note:
Deep dive into search will be discussed in the next parts

- JSON & Elasticsearch -

S

JSON & Elasticsearch

Structuring Your Search

Introduction:

- Elasticsearch uses JSON (JavaScript Object Notation) for configuring and managing its operations, including search queries.

Why JSON:

- A lightweight data interchange format, JSON is easy for humans to read/write and easy for machines to parse/generate.

JSON & Elasticsearch

Basic Components of a JSON Query

Root Object:

- Every Elasticsearch JSON query starts and ends with curly braces { }, defining the root object.

Keys & Values:

- Within the root object, data is organized as key-value pairs, with keys being strings (in double quotes) and values taking various forms (strings, numbers, arrays, or even other objects).

Arrays:

- Values can be lists or arrays, denoted by square brackets [ ].

S

JSON & Elasticsearch

A Simple JSON Query Example

Breakdown:

Root Object:

- The outermost { }.

Key-Value Pairs:

- "query": {...} is a key-value pair where the key is "query" and the value is another object.

Nested Objects:

- Within the "query" object, there's another nested key-value pair "match": {...}.

```
{
  "query": {
    "match": {
      "title": "Inception"
    }
  }
}
```

S

JSON & Elasticsearch

Why This Structure?

Hierarchy & Precision:

- The nested structure allows defining complex and precise queries.

Flexibility:

- Can accommodate simple keyword searches to intricate, multifaceted queries.

Readability:

- Despite the potential complexity, JSON remains readable due to its organized structure.

S

JSON & Elasticsearch

Tips for First-Timers

Formatting:

- Ensure proper placement of commas , and avoid trailing commas.

Validation:

- Use tools like JSONLint to validate your JSON structure if unsure.

Practice:

- The best way to get familiar with the JSON query structure is through hands-on practice and experimentation.

- LABS! -

Lab: Advanced Index Creation

Create multiple indices with various configurations and analyze their settings

Set up three indices:

- \`movielens_text\`, \`movielens_keyword\`, and \`movielens_mixed (moviename need to be keyword AND text)\` where each focuses on a different field type for movie titles. This helps understand the difference between \`text\` and \`keyword\` types.

After creation

- fetch the settings for each index using the \`_settings\` API. Examine the number of shards and replicas for each.

Answer:

- How might the choice between \`text\` and \`keyword\` impact search operations?

Lab: Advanced Single Movie Indexing

Dive deeper into indexing with custom and automatic document IDs.

1. Index the same movie into `movielens` twice: once with a custom document ID and once letting Elasticsearch auto-generate it.

2. Retrieve both documents and observe the differences in their IDs.

3. Answer: What are the implications of choosing manual vs. auto-generated IDs?

Lab: Bulk Indexing with Errors

Understand the effects of erroneous data during bulk indexing.

1. Prepare a bulk insertion of 5 movies, but deliberately introduce a data type mismatch in one of the movies.
2. Use the Bulk API for indexing and observe the response.
3. Identify which movie failed to index and why. Then, correct the error and re-index.
4. Answer: How does Elasticsearch handle individual failures during bulk indexing?

Lab: Modifying Movies & Conflict Handling (skip once we get to python)

Dive into scenarios of concurrent data modifications.

1. Choose a movie and update its title. Immediately after, try deleting the same movie.

2. Observe any version conflict errors and discuss their implications.

3. Using two different REST clients or tabs, try to update the same movie simultaneously and observe conflict behavior.

Lab: Deep Dive into Concurrency (skip)

Understand how Elasticsearch handles concurrent bulk indexing operations.

1. Prepare two different bulk indexing operations that target some of the same movies.
2. Simultaneously run both bulk indexing commands.
3. Investigate any version conflicts or errors and analyze the final state of the indexed movies.
4. Answer: How does Elasticsearch handle concurrent indexing of the same data?

Lab: Advanced Mappings Exploration

Delve into mapping complexities and their effects

1. Index a movie without a predefined mapping and then fetch its mapping using the `_mapping` API

2. Now, create a new index with predefined mappings for the MovieLens dataset.

3. Index the same movie into the newly created index.

4. Compare the two mappings – one from the dynamically created mapping and the other from the predefined one.

5. Try to index a movie into the predefined mapping index with a field not specified in the mapping. Observe the behavior and errors.

6. Discuss: How does Elasticsearch handle fields not present in predefined mappings?

Lab: Handling Mapping Conflicts

Grasp the challenges and solutions when mapping conflicts arise

1. Using the index with dynamic mappings, try to index two different movies. The catch: one movie has a field with data type text, while the other has the same field with data type number.

2. Observe the conflict error provided by Elasticsearch.

3. Resolve the conflict by reindexing the data into a new index with a properly defined mapping.

4. Think about: The importance of consistent data types and the challenges of relying solely on dynamic mapping.

- Introducing the MovieLens Dataset -

What is MovieLens?

A dataset generated by the GroupLens research lab, often used for recommendation algorithms and data analysis.

Components:

- **Movies**: Contains movie titles, genres, and other metadata.
- **Ratings**: User ratings for different movies.
- **Tags**: User-generated tags for movies.
- **Links**: Identifiers that link to external movie databases.

Relevance to Elasticsearch: A diverse dataset like MovieLens is perfect for demonstrating various aspects of indexing, searching, and analytics in Elasticsearch.

https://grouplens.org/datasets/movielens/

- Importing the MovieLens Dataset -

I

Importing The MovieLens DataSet

Setting up Your Data for Exploration

- Connect to your VSCODE IDE by opening: http://YOURREMOTEIP:5001
- Switch to Branch 'labs'
- Go to welcome/ElasticStack/Code/transformation
- Open New shell terminal in vscode

Download dataset:
In the Shell terminal run:

- wget -P /tmp/mleans/ https://jb-workshop.s3.eu-west-1.amazonaws.com/ml-25m.zip
- sudo apt install -y unzip
- unzip /tmp/mleans/ml-25m.zip

I

Importing The MovieLens DataSet

Setting up Your Data for Exploration

Run ‘ python3 csv_to_bulk.py ‘ and click Run:

- When asked for source folder:
  - Type: ./ml-25m/movies.csv
- When asked for target output:
  - Type: ./ml-25m/movies.json
- When asked to provide 3 comma-separated column titles:
  - Type: movieId,title,genres

Importing The MovieLens DataSet

Setting up Your Data for Exploration

Validate file in your shell:

- head ./ml-25m/movies.json

Expected output:

```
{"index": {}}
{"movieId": "1", "title": "Toy Story (1995)", "genres": "Adventure|Animation|Children|Comedy|Fantasy"}
{"index": {}}
{"movieId": "2", "title": "Jumanji (1995)", "genres": "Adventure|Children|Fantasy"}
```

Importing The MovieLens DataSet

Setting up Your Data for Exploration

Bulk Inserting:

Elasticsearch's Bulk API expects a specific format.

- movies.json.json is appropriately formatted for bulk insertion

```
cd ./ml-25m/
curl -u elastic:changeme -X POST "localhost:9200/movielens/_bulk" -H 'Content-Type: application/json' --data-binary "@movies.json"
```

I

Importing The MovieLens DataSet

Setting up Your Data for Exploration

Validation:

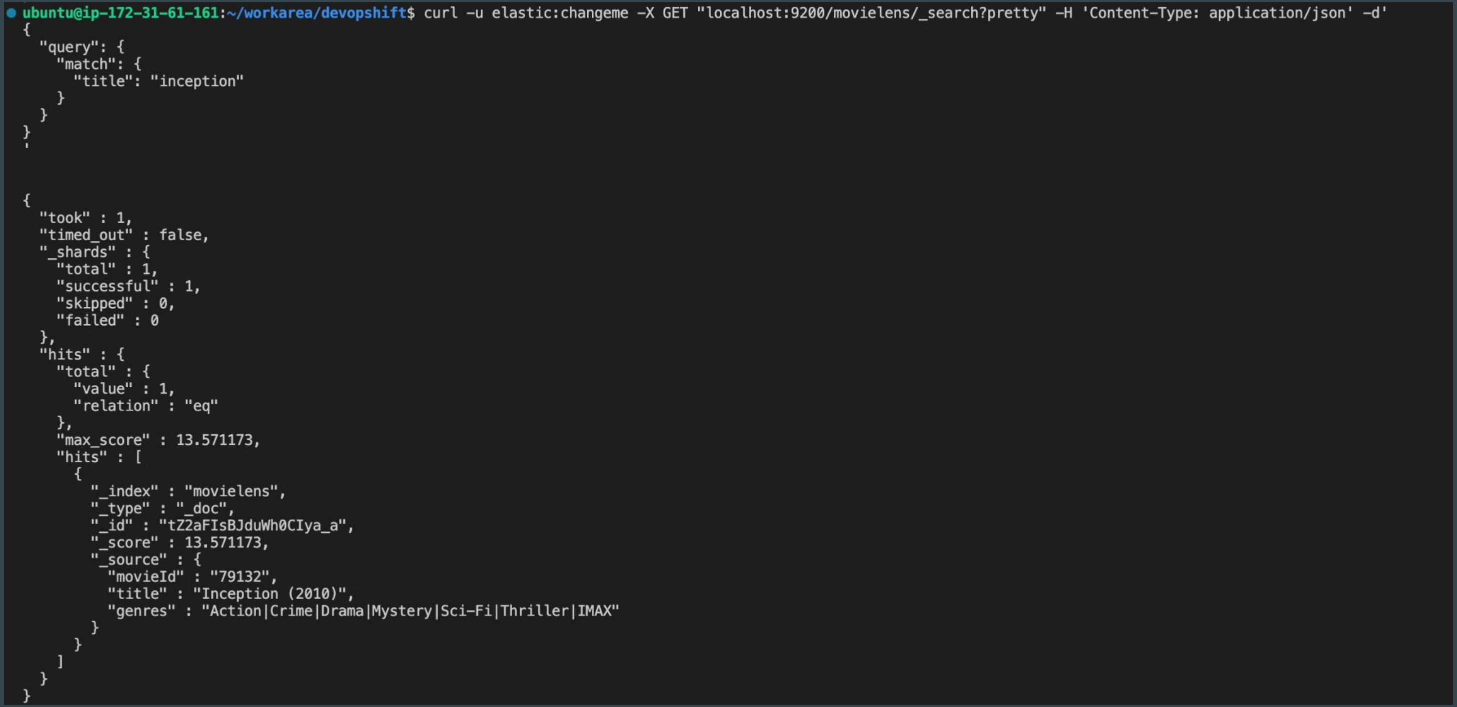
After inserting, a basic search query can validate data insertion:

```
curl -u elastic:changeme -X GET "localhost:9200/movielens/_search?pretty" -H 'Content-Type: application/json' -d'
{
  "query": {
    "match": {
      "title": "inception"
    }
  }
}
'
```

Importing The MovieLens DataSet

Setting up Your Data for Exploration

Expected:



```
ubuntu@ip-172-31-61-161:~/workarea/devopshift$ curl -u elastic:changeme -X GET "localhost:9200/movielens/_search?pretty" -H 'Content-Type: application/json' -d'
{
  "query": {
    "match": {
      "title": "inception"
    }
  }
}
'



{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 13.571173,
    "hits" : [
      {
        "_index" : "movielens",
        "_type" : "_doc",
        "_id" : "tZ2aFIsBJduWh0CIya_a",
        "_score" : 13.571173,
        "_source" : {
          "movieId" : "79132",
          "title" : "Inception (2010)",
          "genres" : "Action|Crime|Drama|Mystery|Sci-Fi|Thriller|IMAX"
        }
      }
    ]
  }
}
```

- BASIC SEARCH LABS -

Lab: Simple Search Operation

Objective: Familiarize yourself with the basic search functionality.

1. Match All Query: Execute a search that fetches all documents from the movielens index.
2. Keyword Search: Look for movies with the title "Matrix".
3. Examine the results. How are they ranked?

Lab: Advanced Search Techniques

Objective: Grapple with the complexities of multi-faceted search operations.

Optional lab: One should try to complete this lab by googling elasticsearch Filters and Sorting

1. Combining Queries: Look for movies that are either in the "Sci-Fi" or "Fantasy" genre and have a rating above 8.
2. Using Arrays: Search for movies that fall into both "Action" and "Adventure" genres.
3. Discussion: Discuss the advantages of Elasticsearch in handling complex queries compared to traditional databases.

NOTE: Filters and Sortings will be cover on the next parts

Lab: Dive Deeper - Filtering & Sorting

Objective: Understand the utility of filtering and sorting in search operations.

Optional lab: One should try to complete this lab by googling elasticsearch Filters and Sorting

1. Filtered Search: Execute a search for movies in the "Action" genre released after 2000.
2. Sorted Results: Retrieve the same results, but sort them by release date in descending order.
3. Answer: Why might sorting and filtering be vital for large datasets?

NOTE: Filters and Sortings will be cover on the next parts

- DEEP DIVE: SEARCH API -

S

Search Types in Elasticsearch

Methods to Query Your Data

- Boolean Search: Combine multiple queries with logical operators (AND, OR, NOT).
- Phrase Search: Search for exact phrases within content.
- Wildcard and Regular Expression Search: Use wildcard characters or regex patterns to match content.
- Fuzzy Search: Search for terms that are similar in spelling.
- Proximity Search: Search for terms that are near each other in content.

Description: Elasticsearch offers a diverse range of query types to fit various search needs. Whether you're looking for exact matches, fuzzy content, or logical combinations, there's a query type tailored for your requirements.

S

Search Types in Elasticsearch

Boolean Search - Combining Queries Logically

Description: The boolean query type combines multiple query types using logical operators. It offers must, should, must_not, and filter clauses

```
{
 "query": {
  "bool": {
   "must": {"match": {"title": "dream"}},
   "filter": {"term": {"year": "2020"}},
   "must_not": {"term": {"genre": "horror"}}
  }
 }
}
```

This fetches documents with the title "dream" from the year 2020, but excludes those with the genre "horror".

S

Search Types in Elasticsearch

Match Boolean Prefix - Broad Prefix Matching

Description: This query type finds documents that match a provided prefix for each term in the query string.

```
{
  "query": {
    "match_bool_prefix": {
      "description": "quick bro"
    }
  }
}
```

This can fetch documents with descriptions like "quick brown", "quick brooding", etc.

S

Search Types in Elasticsearch

Phrase Search - Finding Exact Phrases

Description: Phrase search is ideal when you want to find documents containing an exact sequence of words or terms

```
{
 "query": {
  "match_phrase": {"description": "dream within a dream"}
 }
}
```

This fetches documents with the exact phrase "dream within a dream" in the description.

S

Search Types in Elasticsearch

Match Phrase Prefix - Exact Phrase Prefix Matching

Description: Searches for documents containing a phrase prefix.

```
{
  "query": {
    "match_phrase_prefix": {
      "description": {
        "query": "quick bro"
      }
    }
  }
}
```

Matches documents with phrases starting with "quick bro", like "quick brown dog" or "quick brook".

S

Search Types in Elasticsearch

Wildcard and Regular Expression Search - Flexible Pattern Matching

Description: These searches are powerful when you're uncertain about exact terms or want to match patterns

```
{
  "query": {
    "wildcard": {"title": "dr*am"}
  }
}
```

Matches titles like "dream", "drama", "drum", etc.

```
{
  "query": {
    "regexp": {"name": "dr[ae]m"}
  }
}
```

Matches "dream" and "dram", but not "drum".

Search Types in Elasticsearch

Fuzzy Search - Tolerating Typos

Description: Fuzzy searches use the Levenshtein edit distance to find terms that are similar in spelling, ideal for catching typos or slight misspellings.

```
{
  "query": {
    "fuzzy": {"name": "titanik"}
  }
}
```

This could fetch documents containing the word "titanic".

S

Search Types in Elasticsearch

Proximity Search - Nearby Terms

Description: Proximity searches locate documents where the terms appear within a specified distance from one another

```
{
 "query": {
  "match_phrase": {
   "description": {
    "query": "quick dog",
    "slop": 1
   }
  }
 }
}
```

This could fetch documents like "quick brown dog", but not "quick little brown dog" due to slop 1

S

Search Types in Elasticsearch

Understanding slop - Flexibility in Phrase Matches

Description: In a phrase query, the slop parameter defines how far apart terms are allowed to be while still considering the document a match. A slop of 2, for instance, allows terms to have two positions apart and still match.

```
{
 "query": {
  "match_phrase": {
   "description": {
    "query": "quick dog",
    "slop": 2
   }
  }
 }
}
```

This might match a document with the description "quick little brown dog".

Search Types in Elasticsearch

Proximity Search - Nearby Terms

Description: Proximity searches locate documents where the terms appear within a specified distance from one another

```
{
 "query": {
  "match_phrase": {
   "description": {
    "query": "quick dog",
    "slop": 1
   }
  }
 }
}
```

This could fetch documents like "quick brown dog", but not "quick little brown dog" due to slop 1

Search Types in Elasticsearch

Combined Fields - Searching Across Fields

Description: Allows you to search multiple fields as if they were one.

```
{
 "query": {
  "combined_fields": {
   "query": "quick brown",
   "fields": ["title", "description"]
  }
 }
}
```

This searches both the "title" and "description" fields for the terms "quick" and "brown".

Search Types in Elasticsearch

Multi-match Query - Multi-field Matching

Description: Offers a way to run a match query on multiple fields. Can be particularly useful when searching across fields with different analyzer settings.

```
{
 "query": {
  "multi_match": {
   "query": "quick foxes",
   "fields": ["title", "description"]
  }
 }
}
```

This searches both for “quick foxes” in both title and description fields

- LABS! -

Lab 1: Multi-Field Movie Search

Objective: Craft a search query that leverages the multi-match capability.

Instructions:

1. Using the MovieLens dataset, create a search query that looks for movies with the term "love" in both the title and the description.
2. Adjust your query to boost results where the term appears in the title (show how to use boost)
    a. Compare results. Note how boosting affects the order of returned results.

S

Proximity Search for Descriptions

Objective: Understand the effects of the slop parameter in phrase queries.

Instructions:

1. Search for the phrase "science fiction" in movie descriptions.
2. Adjust the slop parameter to 2 and re-run your search. Note the additional matches that might not be exact phrase matches.
3. Discuss: In what scenarios might allowing for some flexibility in phrase matches be beneficial?

S

Lab 3: Advanced Combined Field Search

Objective: Utilize the combined fields query to construct a more complex search.

Instructions:

1. Search for movies that contain terms "war" and "heroic" across both title and genre.
2. Try to boost the relevance of matches in the title over genre.
3. Examine the returned results. Can you spot movies where "war" appears in the title, but "heroic" might be implied through the genre or vice-versa?
4. Bonus: Combine the above with a range filter to find movies released after the year 2000.

- SOLUTIONS -

S

Lab 1: Multi-Field Movie Search

Objective: Craft a search query that leverages the multi-match capability.

```
{
 "query": {
  "multi_match": {
   "query": "love",
   "fields": ["title", "description"]
  }
 }
}
```

S

Proximity Search for Descriptions

Objective: Understand the effects of the slop parameter in phrase queries.

```
{
 "query": {
   "match_phrase": {
     "description": {
       "query": "science fiction",
       "slop": 2
     }
   }
 }
}
```

S

Lab 3: Advanced Combined Field Search

Objective: Utilize the combined fields query to construct a more complex search.

```
{
  "query": {
    "bool": {
      "must": [
        {
          "combined_fields": {
            "query": "war heroic",
            "fields": ["title^2", "genre"]
          }
        },
        {
          "range": {
            "release_year": {
              "gte": 2000
            }
          }
        }
      ]
    }
  }
}
```

- Filter Search Results -

Search Types in Elasticsearch

Enhancing Precision in Search

Description: Filtering offers a way to further refine search results without affecting the relevancy score. Filters provide binary (yes/no) decisions on whether a document matches, allowing for efficient, cacheable operations

```
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "title": "love"
        }
      },
      "filter": {
        "range": {
          "release_year": {
            "gte": 2000
          } ....
```

Structure:

- **Post-Filter**: Applied after the initial query, suitable for cases like "top N" filtering.
- **Filter Context**: Typically used within the bool query, it's **more cacheable** and hence faster.

This query searches for movies with "love" in the title but filters results to only those released after the year 2000.

S

Search Types in Elasticsearch

Filter Search Results - Deep Dive into Filtering Options

Description: Elasticsearch provides a plethora of filtering options to enhance the precision and relevancy of your search results

Search Types in Elasticsearch

Filter Search Results - Deep Dive into Filtering Options

Structure:

Range Filters:

- gte (Greater Than or Equal)
- lte (Less Than or Equal)
- gt (Greater Than)
- lt (Less Than)

Term & Terms Filter:

- To filter results matching a particular term or multiple terms.

Exists & Missing Filter:

- Check for the existence or non-existence of a field.

Bool Filter:

- Combines multiple filter criteria

Search Types in Elasticsearch

Filter Search Results - Deep Dive into Filtering Options

EXAMPLE:

This filters movies with the title containing "adventure" and having a rating between 4 and 5 (inclusive).

```
{
 "query": {
  "bool": {
   "must": {
    "match": {
     "title": "adventure"
    }
   },
   "filter": {
    "range": {
     "rating": {
      "gte": 4,
      "lte": 5
     }
    }
   }
  }
 }
}
```

S

Search Types in Elasticsearch

Highlighting - Visual Clues for Matches

Description: Elasticsearch provides a way to highlight the search matches within the text of the results, aiding in visual perception of relevance.

```
{
 "query": {
   "match": {
     "description": "hero"
   }
 },
 "highlight": {
   "fields": {
     "description": {}
   },
   "pre_tags": ["<strong>"],
   "post_tags": ["</strong>"]
 }
```

Structure:

- **Pre-Tags & Post-Tags:** Define the HTML tags (or other symbols) that will wrap the highlighted terms.
- **Fields**: Specify which fields to highlight.

This query searches for the term "hero" in movie descriptions and highlights the matches by wrapping them with <strong> tags.

S

Search Types in Elasticsearch

Highlighting - Advanced Usage - Emphasizing Matches for Better Visualization

Basic: Highlighting a term in a field.

```
{
  "query": {
    "match": {
      "description": "adventure"
    }
  },
  "highlight": {
    "fields": {
      "description": {}
    },
    "pre_tags": ["<strong>"],
    "post_tags": ["</strong>"]
  }
}
```

Output:

```
"hits": {
  "total": {...},
  "max_score": ...,
  "hits": [
    {
      "_index": "...",
      "_type": "...",
      "_id": "...",
      "_score": ...,
      "highlight": {
        "description": [
          "Join this epic <strong>adventure</strong> in the mountains."
        ]...
```

Search Types in Elasticsearch

Highlighting - Advanced Usage - Multiple Fields Highlighting

Basic: Highlighting a term in a field.

```
{
 "query": {
   "multi_match": {
     "query": "hero",
     "fields": ["title", "description"]
   }
 },
 "highlight": {
   "fields": {
     "title": {},
     "description": {}
   },
   "pre_tags": ["<em>"],
   "post_tags": ["</em>"]
 }
}
```

Output:

```
"hits": {
......
   "highlight": {
     "title": [
       "<em>Hero</em> of the Desert"
     ],
     "description": [
       "A true story of a <em>hero</em> who fought against all odds."
     ]
   }
  }
 ]
}
```

S

Search Types in Elasticsearch

Using Different **Highlight** Tags: Providing varied visual emphasis

```
{
  "query": {
    "match": {
      "description": "journey"
    }
  },
  "highlight": {
    "fields": {
      "description": {}
    },
    "pre_tags": ["<mark>"],
    "post_tags": ["</mark>"]
  }
}
```

Output:

```
"hits": {
......
    {
      "_index": "...",
      "_type": "...",
      "_id": "...",
      "_score": ...,
      "highlight": {
        "description": [
          "The <mark>journey</mark> began at dawn."
        ]
      }
    }
....
```

- Understanding Query Caching in Elasticsearch -

Understanding Query Caching in Elasticsearch

Boosting Performance by Caching

Description: Elasticsearch optimizes the performance of repeated queries through caching. It saves the results of expensive filter operations for reuse.

Structure:

- **Shard-Level Caching**: Cache is maintained at the shard level.
- **Filter Context**: Filters in the filter context are cacheable.
- **Cache Eviction**: Cache is cleared upon any change in the mapping or data to ensure result accuracy.

Benefits:

- Improved response times for frequent queries.
- Reduced CPU usage.

Note: While caching boosts performance,
it's crucial to balance between caching and the freshness of data. Regularly accessed filters benefit the most from caching.

Understanding Query Caching in Elasticsearch

Query Caching - Activation & Maintenance

Description: Ensuring Quick Responses without Sacrificing Data Freshness

Activation:
By default, query caching is active for certain types of queries (those executed in the filter context).
However, you can explicitly control it -

S

Understanding Query Caching in Elasticsearch

Query Caching - Activation & Maintenance

Enable/Disable Cache at Query Level:

```
{
  "query": {
    "bool": {
      "filter": {
        "term": {
          "user": "john",
          "_cache": true  // Explicitly activate caching for this query.
        }
      }
    }
  }
}
```

Understanding Query Caching in Elasticsearch

Query Caching - Activation & Maintenance

Description: Ensuring Freshness
Elasticsearch automatically clears the cache on the relevant shards whenever changes are made to the data or mapping. This means every indexing, delete, update, or bulk request.

However, it's essential to be aware:

- Frequent updates can cause the cache to be continuously cleared, leading to reduced cache benefits.

- If you have infrequent data changes but frequent queries, cache efficiency improves.

S

Understanding Query Caching in Elasticsearch

Monitoring & Management

Description: Elasticsearch provides API endpoints to inspect and manage the cache:

View Cache Statistics:

```
GET /_nodes/stats/indices/query_cache
```

Clear Cache:
For specific index:

```
POST /<index_name>/_cache/clear
```

Clear Cache:
For all indices:

```
POST /<index_name>/_cache/clear
```

Understanding Query Caching in Elasticsearch

Shard-Level Caching vs. Node-Level Caching

Description: The shard-level cache is particularly useful since, during a search operation, each shard can quickly check its cache to see if it already contains the result for a filter.

Note:
While caching improves query speed, over-reliance or not correctly managing it can lead to increased memory usage. Regularly monitoring cache stats is crucial to ensure an optimized balance between speed and resource consumption.

Advanced monitoring - TBD

- DEEP DIVE SEARCH CONTINUE -

Search Types in Elasticsearch

Long-running Searches - The Scroll API - Ensuring Complete Data Retrieval for Intensive Queries

Description: Elasticsearch is designed to be fast and to return data in near real-time. For most use-cases, queries return in milliseconds. However, when dealing with a massive volume of data and when retrieval of a significant chunk of that data is required, standard search queries might time out or affect cluster performance.

For such scenarios, Elasticsearch provides the Scroll API.

Search Types in Elasticsearch

Long-running Searches - The Scroll API - Ensuring Complete Data Retrieval for Intensive Queries

Basic Concept of Scroll API:

1. Basic Concept of Scroll API:

The Scroll API takes a snapshot of the current state of an index when the search starts. This ensures consistency across returned pages, even if data changes during the search.

Search Types in Elasticsearch

Long-running Searches - The Scroll API - Ensuring Complete Data Retrieval for Intensive Queries

Basic Concept of Scroll API:

Starting a Scroll Session

- To initiate a long-running search, use the scroll parameter in your search request:

```
GET /movielens/_search?scroll=1m
{
  "query": {
    "match_all": {}
  },
  "size": 100  // Number of results returned per "page"
}
```

Here, 1m ensures the search context remains alive for 1 minute. Adjust this based on expected search duration.

S

Search Types in Elasticsearch

Long-running Searches - The Scroll API - Ensuring Complete Data Retrieval for Intensive Queries

Basic Concept of Scroll API:

Fetching Subsequent "Pages":

- After the initial search, you'll receive a scroll_id. Use this ID to retrieve the next batch:

```
GET /_search/scroll
{
  "scroll" : "1m",
  "scroll_id" : "DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAAD4WYm9laWdfTWpDZ05USXliSUZLMmJwUDJBZz0zRA=="
}
```

S

Search Types in Elasticsearch

Long-running Searches - The Scroll API - Ensuring Complete Data Retrieval for Intensive Queries

Basic Concept of Scroll API:

Closing the Scroll Session

- After you retrieve all required data, it's essential to close the scroll context to free up resources:\

```
DELETE /_search/scroll
{
  "scroll_id": [
    "DXF1ZXJ5QW5kRmV0Y2gBAAAAAAAAD4WYm9laWdfTWpDZ05USXliSUZLMmJwUDJBZz0zRA=="
  ]
}
```

Note: While the Scroll API ensures consistency and allows for long-running searches, it's not intended for real-time user requests but rather for processes like data exports.

P

Paginating Search Results - From & Size

Efficiently Navigating Through Large Result Sets

Basic Concept of Pagination

Elasticsearch by default returns 10 search hits. To return more or fewer hits or to paginate results, you can use the from and size parameters.

P

Paginating Search Results - From & Size

Basic Pagination Example

Retrieve results 11 to 20

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

Output

```
{
  "hits": {
    "total": { "value": 10000 },
    ...
    // 10 hits starting from the 11th record.
  }
}
```

P

Paginating Search Results - From & Size

Limitations & Considerations

The from + size values can't exceed the index.max_result_window index setting (default to 10,000).
For retrieving vast amounts of data, consider Search After API (TBD).

P

Paginating Search Results - Using search_after

Efficiently Navigating Through Large Result Sets Beyond the First 10,000 Hits

Basic Concept of search_after:
For deep pagination beyond the first 10,000 results, Elasticsearch recommends using the search_after parameter combined with a Point In Time (PIT).

Starting a PIT session

```
POST /movielens/_pit?keep_alive=5m
```

This initiates a PIT session for 5 minutes. The response will provide a pit_id which you'll use in subsequent searches.

Paginating Search Results - Using search_after

Using search_after with PIT

To paginate, we typically sort the results by some field(s).
Then, in the next query, we provide the last retrieved values of those fields to fetch the subsequent page.

```
GET /movielens/_search
{
  "pit": {
    "id": "YOUR_PIT_ID_HERE"
  },
  "query": {
    "match_all": {}
  },
  "sort": [
    {"timestamp": "asc"}
  ],
  "size": 10,
  "search_after": ["TIMESTAMP_OF_LAST_HIT"]
}
```

Replace **YOUR_PIT_ID_HERE** with the **PIT ID** from the previous request and **TIMESTAMP_OF_LAST_HIT** with the timestamp value of the last hit from the previous page.

Paginating Search Results - Using search_after

Limitations & Considerations

While search_after provides a solution for deep pagination, it's essential to close the PIT session after you're done to free up resources:

```
DELETE /_pit
{
  "id": ["YOUR_PIT_ID_HERE"]
}
```

search_after is not suited for real-time user requests but more for batch jobs or exporting data.

Retrieve Selected Fields

Efficiently Fetching Only the Required Data

Basic Concepts To optimize search performance and reduce network overhead, Elasticsearch provides a way to retrieve only specific fields from documents instead of the complete source.

R

Retrieve Selected Fields

Efficiently Fetching Only the Required Data

Using _source Filtering
You can define which fields to retrieve using the **_source** parameter:

```
GET /movielens/_search
{
  "_source": ["title", "release_date"],
  "query": {
    "match": { "title": "Inception" }
  }
}
```

Output:

```
{
  "hits": {
    "hits": [
      {
        "_source": {
          "title": "Inception",
          "release_date": "2010-07-16"
        }...
```

R

Retrieve Selected Fields

Efficiently Fetching Only the Required Data

Benefits:

- Reduce network overhead.
- Improve search performance, especially with large documents.
- Fetch only the data that is required for a specific use-case.

S

Search Multiple Data Streams and Indices

Broadening the Search Scope

Multi-Index Search

Elasticsearch allows you to search across multiple indices with a single request:

```
GET /movielens,moviearchive/_search
{
 "query": {
  "match": { "title": "Inception" }
 }
}
```

This query searches both the movielens and moviearchive indices for movies titled "Inception."

S

Search Multiple Data Streams and Indices

Broadening the Search Scope

Using Wildcards

You can also use wildcards in the index name to match multiple indices:

```
GET /movie*/_search
{
  "query": {
    "match": { "title": "Inception" }
  }
}
```

Benefits & Considerations:

- Broaden the search scope without making multiple requests.
- Useful when data is distributed across multiple indices or when unsure which index contains the required information.
- Ensure you're not overloading the search by querying too many unnecessary indices.

Sort Search Results

Ordering the Search Output

Default Sorting

By default, Elasticsearch sorts results by relevance score.

Custom Sorting

You can specify one or more fields to sort by:

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

Combining with Filters

You can also combine sorting with filters:

```
GET /movielens/_search
{
  "query": {
    "term": { "genre": "Action" }
  },
  "sort": [
    { "rating": "desc" }
  ]
}
```

Searching with Query Rules

Boosting, Must, Must Not and More

Boolean Queries
Elasticsearch's Bool query combines multiple query criteria (must, must_not, should) with boosting.

Example
Finding action movies but not those titled "Die Hard" and boosting newer movies:

```
GET /movielens/_search
{
  "query": {
    "bool": {
      "must": {
        "term": { "genre": "Action" }
      },
      "must_not": {
        "term": { "title": "Die Hard" }
      },
      "should": {
        "range": { "release_date": { "gte": "2015-01-01" } }
      },
      "boost": 2.0
    }
  }
}
```

- AUTOCOMPLETE -

Autocomplete Mechanisms in Elasticsearch

Different Strategies for User-friendly Search

Prefix Query:

- Simplest form of autocomplete.
- Matches terms that begin with the specified prefix.
- Example: {"query": {"prefix": {"title": "sta"}}} will match "Star Wars", "Stargate", etc.

Edge Ngram Tokenizer

- Splits words into subtokens.
- E.g., "dog" becomes ["d", "do", "dog"].
- Great for partial word matches but can increase index size.

Completion Suggester

- Specifically built for autocompletion.
- Uses a data structure (FST) for fast lookups.
- Stores potential completions as a separate field.

A

Autocomplete Mechanisms in Elasticsearch

Working with Edge Ngram Tokenizer - Building Partial Word Matches - INTRO

Concept

- As text is indexed, it's split into subtokens.
- E.g., "autocomplete" becomes ["a", "au", "aut", "auto", ...].

Implementation:

- Define a custom analyzer that uses the edge ngram tokenizer.
- Apply this analyzer to the desired field.

Example:

- Tokenizing "autocomplete" will match on "auto", "autoc", "autoco" and so on.

A

Autocomplete Mechanisms in Elasticsearch

Leveraging the Completion Suggester - Optimal Autocomplete with Elasticsearch - Intro

Why Use It?

- Built specifically for autocomplete.
- Efficient, especially for large datasets.

Setting It Up:

- Add a completion type field to mappings.
- Index the data into this field.

Querying:

- Use the _search endpoint with the suggest construct.
- Elasticsearch returns potential completions based on the input.

A

Autocomplete Mechanisms in Elasticsearch

Fine-tuning with the "slop" Parameter - Flexibility in Phrase Matching - Intro

What's Slop?

- Allows for matching phrases with intervening words.
- E.g., "quick brown fox" with a slop of 1 could match "quick fox".

Usage:

- Used with the "match_phrase" query type.

Example:

- Query for "quick fox" with a slop of 1 will match documents containing "quick brown fox".

A

Autocomplete Mechanisms in Elasticsearch

Edge Ngram Tokenizer - Tokenizing Text for Prefix Matching

What is Edge Ngram Tokenizer?

- A tokenizer that divides text into substrings, based on character patterns.
- Unlike the standard tokenizer, it can divide a word into smaller chunks for partial matching.

Usage:

- Commonly used for autocomplete and search-as-you-type functionality.
- For example, "autocomplete" gets tokenized as ["a", "au", "aut", "auto", ...].

Configuration:

- Can be tailored to define minimum and maximum gram lengths.
- Example: A configuration with min_gram: 2 and max_gram: 3 will tokenize "dog" into ["do", "dog"].

A

Autocomplete Mechanisms in Elasticsearch

Edge Ngram Tokenizer - Implementation - INTRO

Analyzer Configuration:

- Create a custom analyzer that uses the edge ngram tokenizer.

Applying to Mappings:

- Add or update the desired field to utilize the custom analyzer for indexing.
- "title": { "type": "text", "analyzer": "edge_ngram_analyzer" }

```
{
 "settings": {
  "analysis": {
   "tokenizer": {
    "edge_ngram_tokenizer": {
     "type": "edge_ngram",
     "min_gram": 1,
     "max_gram": 25,
     "token_chars": ["letter"]
    }
   },
   "analyzer": {
    "edge_ngram_analyzer": {
     "type": "custom",
     "tokenizer": "edge_ngram_tokenizer"
    }
   }
  }
 }
}
```

- Step-by-step Solution for Building Custom Autocomplete -

A

Autocomplete Mechanisms in Elasticsearch

Step-by-step Solution for Building Custom Autocomplete

Step 1: Update the MovieLens Index to Incorporate the Edge Ngram Tokenizer
Firstly, you'd need to either update the index settings or create a new index with the desired settings.
As direct updating of existing tokenizer settings on an index is not possible, we will create a new index:

A

Autocomplete Mechanisms in Elasticsearch

movielens_v2

```
PUT /movielens_v2
{
 "settings": {
  "analysis": {
   "tokenizer": {
    "edge_ngram_tokenizer": {
     "type": "edge_ngram",
     "min_gram": 1,
     "max_gram": 25,
     "token_chars": ["letter"]
    }
   },
   "analyzer": {
    "edge_ngram_analyzer": {
     "type": "custom",
     "tokenizer": "edge_ngram_tokenizer"
    }
   }
  }
 },
 "mappings": {
  "properties": {
   "title": {
    "type": "text",
    "analyzer": "edge_ngram_analyzer"
   }

  }
 }
}
```

A

Autocomplete Mechanisms in Elasticsearch

movielens_v2

Step 2: Reindex Existing Movie Titles to Apply the New Tokenizer

Now that we have a new index with the desired settings, we should reindex our data from the old index to this new one:

```
POST _reindex
{
  "source": {
    "index": "movielens"
  },
  "dest": {
    "index": "movielens_v2"
  }
}
```

This will transfer all data from movielens to movielens_v2 while applying the new settings and mappings.

A

Autocomplete Mechanisms in Elasticsearch
movielens_v2

Step 3: Query with Edge Ngram Analyzer for Autocomplete
Now you can query the movielens_v2 index to get autocomplete suggestions. For example:

```
GET movielens_v2/_search
{
 "query": {
  "match": {
   "title": {
    "query": "sta",
    "analyzer": "edge_ngram_analyzer"
   }
  }
 }
}
```

This query should provide movies that start with "sta", e.g., "Star Wars", "Stargate", etc.

A

Autocomplete Mechanisms in Elasticsearch

Movielens_v2 - update our server.py to use the new index and ngram

In our server.py update the search route and definition

```
@app.route('/search', methods=['POST'])
def search():
    keyword = request.form.get('keyword')
    body = {
        "query": {
            "match": {
                "title.edge_ngram": keyword
            }
        },
        "_source": ["title"],  # to return only the movie title
        "size": 10
    }
    response = es.search(index="movielens_v2", body=body)  # using the new index with edge ngram settings
    return jsonify(response['hits']['hits'])
```

This code integrates the edge ngram tokenizer by querying against the title.edge_ngram field in the movielens_v2 index. The search results should return titles that are autocompleted based on the user's input.

- Update the index mapping for Completion Suggester -

A

Completion Suggester

specialized tool for autocomplete in Elasticsearch - **Perfect option**

First, we'll define a new index or update the mapping of an existing one to include the completion field.

```
PUT /movielens_v3
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text"
      },
      "title_suggest": {
        "type": "completion"
      }
    }
  }
}
```

Secondly, we'll reindex our data using script option (TBD) to populate the title_suggest field

```
POST /_reindex
{
  "source": {
    "index": "movielens"
  },
  "dest": {
    "index": "movielens_v3"
  },
  "script": {
    "source": "ctx._source.title_suggest = ['input': ctx._source.title]"
  }
}
```

A

Completion Suggester

Test it:

Using the suggest functionality to get movie title suggestions based on a given prefix:

```
POST movielens_v3/_search
{
 "suggest": {
  "movie-title-suggestion" : {
   "prefix" : "inter",
   "completion" : {
    "field" : "title_suggest"
   }
  }
 }
}
```

In this example:

- The prefix "inter" is used as the starting characters of the movie title you want suggestions for.
- title_suggest is the field defined as a completion type.
- This query will return movie titles starting with "inter" as suggestions. The suggestions are returned in the suggest section of the response.

A

Autocomplete Mechanisms in Elasticsearch

Movielens_v2 - update our server.py to use the new index and ngram

In our server.py update the search route and definition

```
@app.route('/search', methods=['POST'])
def search():
    keyword = request.form.get('keyword')
    body = {
        "suggest": {
            "movie-suggest": {
                "prefix": keyword,
                "completion": {
                    "field": "title_suggest"
                }
            }
        }
    }
    response = es.search(index="movielens_v3", body=body)  # using the new index with completion suggester
    suggestions = response['suggest']['movie-suggest'][0]['options']
    return jsonify(suggestions)
```

With this setup, when you enter a query in the search interface, Elasticsearch will provide autocompletion suggestions based on the title_suggest field in the movielens_v3 index.

- Data Relationships -

- Normalization vs. Denormalization - Introduction -

A

Normalization vs. Denormalization - Introduction

Understanding Data Structures in Elasticsearch

Description: Before diving into how Elasticsearch handles data, it's crucial to grasp the foundational concepts of data structuring: normalization and denormalization.

A

Normalization vs. Denormalization - Introduction

What is Normalization?

Definition:

- The process of efficiently organizing data in databases.

Goal:

- Minimize redundancy and dependency by organizing fields and table of data.

Use-Cases:

- Relational databases where data integrity and ACID (atomicity, consistency, isolation, durability) properties are vital.

Example:

- Splitting a table into two tables to eliminate data redundancy.

A

Normalization vs. Denormalization - Introduction

What is Denormalization?

Definition:

- The process of integrating normalized datasets for querying and analytical purposes.

Goal:

- Improve performance by adding redundant copies of data

Use-Cases:

- Data warehousing, NoSQL databases, and Elasticsearch for faster read operations.

Example:

- Combining multiple tables into one table/view to avoid joins

A

Normalization vs. Denormalization - Introduction

Trade-offs -

Normalization

Pros:

- Data integrity, minimizes redundancy, smaller data size.

Cons:

- Increased complexity, potentially slower read performance due to joins.

Denormalization

Pros:

- Faster reads, simpler query patterns, no joins needed.

Cons:

- Potential for data inconsistency, larger data size, complex write operations

A

Normalization vs. Denormalization - Introduction

Elasticsearch's Approach

Primarily Denormalized:

- Elasticsearch, as a search engine, is optimized for read-heavy operations.
- Nested Objects & Parent-Child: While denormalization is preferred, ES offers ways to maintain relationships.

Example: E-commerce Products

- **Normalized** (Relational DBs): Separate tables for products, categories, and manufacturers.
- **Denormalized** (Elasticsearch): A single document for a product containing all relevant information, including category details and manufacturer data.

A

Normalization vs. Denormalization - Introduction

Strategy in Elasticsearch

Default to Denormalization:

- Given its nature, start with a denormalized approach.

Consider Relationships:

- Use tools like nested objects and parent-child only when necessary to maintain data relationships (TBD)

Evaluate Use Case:

- Understand the specific requirements of your application before deciding on a data model.

- Handling Arrays in Elasticsearch -

A

Handling Arrays in Elasticsearch

Understanding and Working with Arrays

Description: In Elasticsearch, any field can contain zero or more values by default. However, there is no dedicated array data type. A field's data type is determined by the type of the first value you index

Handling Arrays in Elasticsearch

Basic Concepts

No Special Array Type:

- Unlike some other systems, you don't explicitly define a field as an array. If you provide multiple values, Elasticsearch recognizes it as an array.

Same Data Type:

- All values in the array must be of the same data type. You cannot mix, for instance, strings and numbers in the same array.

A

Handling Arrays in Elasticsearch

Indexing Arrays

Example Document:

```
{
  "movie": "Inception",
  "tags": ["sci-fi", "thriller", "dream"]
}
```

Description: The tags field is recognized as an array because it holds multiple string values.

A

Handling Arrays in Elasticsearch

Searching in Arrays

Simple Queries: Searching arrays is as straightforward as searching single-value fields.

```
{
  "query": {
    "match": {
      "tags": "dream"
    }
  }
}
```

Exact Match: If you're looking for documents where an array field contains multiple specific values (e.g., both "sci-fi" and "dream"), you can use the terms query.

```
{
  "query": {
    "terms": {
      "tags": ["sci-fi", "dream"]
    }
  }
}
```

A

Handling Arrays in Elasticsearch
Multi-Value Fields vs. Nested Objects

Multi-Value Fields:

- As demonstrated, Elasticsearch handles arrays as multi-value fields. These are flat structures.

Nested Objects:

- If you have arrays of objects and each object needs to be indexed and queried as an independent document, you'd use the nested data type.

TBD - Nested Objects

- Retrieve Inner Hits -

R

Retrieve Inner Hits

Fetching Relevant Nested or Parent/Child Documents

Basic Concept of Inner Hits:

Inner hits return per search hit in the search response nested or parent/child related inner hits.
This is particularly useful when dealing with parent-child relationships or nested documents.

R

Retrieve Inner Hits

Fetching Relevant Nested or Parent/Child Documents

Document with Nested Comments Example:
Consider that each movie can have nested comments. First, let's establish a sample document structure for a movie with nested comments:

```
PUT /movielens/_doc/1
{
  "title": "Inception",
  "release_date": "2010-07-16",
  "comments": [
    { "user": "John", "text": "Amazing movie!" },
    { "user": "Alice", "text": "I was confused at times, but still
good." }
  ]
}
```

R

Retrieve Inner Hits

Fetching Relevant Nested or Parent/Child Documents

Querying Nested Documents:
To search for comments containing the word "Amazing" and retrieve the corresponding nested documents:

```
GET /movielens/_search
{
  "query": {
    "nested": {
      "path": "comments",
      "query": {
        "match": {
          "comments.text": "Amazing"
        }
      },
      "inner_hits": {}  // This fetches the matching nested
documents.
    }
  }
}
```

R

Retrieve Inner Hits

Fetching Relevant Nested or Parent/Child Documents

Output:

```
{
 "hits": {
  "total": { "value": 1 },
  "hits": [
   {
    "_source": { "title": "Inception", ... },
    "inner_hits": {
     "comments": {
      "hits": {
       "total": { "value": 1 },
       "hits": [
        { "_source": { "user": "John", "text": "Amazing movie!" } }
       ]
      }
     }
    }
   }
  ]
 }
}
```

Practical Uses of Inner Hits:

- Retrieving specific comments or reviews related to a product or movie.
- Fetching relevant sections of a larger document.
- Isolating related nested objects within a larger structured dataset.