Indexing and Searching in Elasticsearch

What is an elasticsearch index and how does it work?

An ***elasticsearch index*** is the place or the location to store and organize the data which we load to elasticsearch. The equivalent to an elasticsearch index in the relational database world is a database which contains multiple tables. There is a small difference in the two, where in a relational database, a single table is designed to store a single type of data. Where as an elasticsearch index can store data but all of them don't need to be the same, but they generally have to be related to one another (departments / employees).

The data records which we load / store in an elasticsearch index is called a ***document***. Documents are the things we search for after indexing / loading. A document can be more than just text and it could be any structured JSON object. Each document which gets stored in elasticsearch index will have a unique ID, which is unique to the index which its being loaded into. Additionally each document will have a type as well.

The indices in elasticsearch are actually ***inverted indices**.

What is an ***inverted index***

Let's take two documents (eg. example data records) which we want to store in an elasticsearch index to search for.

Inverted Index in Elasticsearch

As we can see in the above diagram, the documents when they get indexed into the index, elasticsearch analyzes the document, and tokenizers the document then keeps the total count of the tokens (which is equal to a word in this example).

An index is made up or one or more primary shards. Then for resiliency the primary shards are replicated to different elasticsearach hosts to make replica shards. Below we can see how the index is split into multiple primary shards. This is mainly done to improve the write performance into an elasticsearch shards. As a best practice the primary shards are spread across multiple elastcsearch hosts to improve the write performance.

Index Splitted into multiple Primary Shards

The below figure, shows how each primary shard is replicated into other elastisearch nodes to create replica shards. This is done mainly to improve the data resiliency in case of a elasticsearch node failure. Replica shards are mainly used for read requests (searching requests from users / apps).

Primary Shards and Replica Shards

The number of primary shards and replica shards can be configured. The primary shards count has to be defined at the time we create the index and not allowed to change one defined. The only way to change the number of primary shards is by reindexing the data into a new index.

The number of replica shards can be changed dynamically to improve the read performance and hence it supports increasing and decreasing the replica shard count. But changing replica shards has a huge compute power and storage space impact, therefore careful thought needs to be given before doing that.

```
PUT /test-index
{
    "settings": {
        "number_of_shards": 3,
        "number_of_replicas": 1
    }
}
```

Elasticsearch in the backend tries to evenly distribute the data across the primary shards to keep the performance of each primary shard in optimum level.

Indexing is the action of writing documents into an elasticsearch index. We can create an index using the create index API.

Let us get our hands dirty now.

> Tips
>
> **Using Kibana Dev Tools**
>
> Eleasticsearch Kibana interface provides a very usefull tool for us to build and test queries. Rather than using any other tools like curl or Postman to invoke the elasticsearch apis.
>
> Using the main Kibana Menu we can select Dev Tools under Section - Management
>
> Kibana Dev tools for Query building and testing
>
> Below is the actual Dev Tools interface
>
> Kibana Dev Tools interface

Let's log into the Kibana Dev Tools and check for the health of our elasticsearch cluster.

```
# Querying for Elastic Cluster Health
GET _cluster/health
```

The status - "yellow" is because we are running on a single node, and recommended minimum cluster size is 3 nodes. Don't worry about that for now. We can see that *number_of_nodes* is 1 and *number_of_data_nodes* is again 1. You can see the number of primary shards and many more parameters which indicates the overall health of the cluster.

Let us now create an index and get its details. We can create an index with a PUT request.

```
# Creating the index
PUT my-index

# Getting the details of the index
GET my-index
```

We can see that the index named my-index was created, and the default was to have a single primary shard with 1 replica. The *mapping* section is empty. The *mapping* is the definition of the data structure or the schema (referring to the relational database terminology) of the documents which we store in the index. That's because we have not indexed any documents into the index yet.

We can get a list of all the indices in the cluster as below.

```
# Getting details of the current indices
GET _cat/indices
```

Let us know create an index with 3 primary shards and 1 replica shard.

```
PUT my-other-index
{
  "settings": {
    "index": {
      "number_of_shards": 3,
      "number_of_replicas": 1
    }
  }
}
```

> Tips
>
> The data in the shards should be evenly distributed for best performance. We have seen expert advice given to keep the shard size to 30GB - 50GB for high performing search use cases.

Inside of an Index

As mentioned earlier, a data record which we store in an index is called ***document*** and the equivalent of this from the relational database jargon is a record in a table. The content gets indexed in elasticsearch as a JSON object.

JSON documents are complex data structures and contains ***key / value pairs***. They ***keys*** are generally strings, while ***values*** can be *nested objects, arrays, or data types such as datetime, geo_points, IP addresses* and more.

Index Settings

Index attributes and functional parameters can be defined using index settings. The settings can be *dynamic* as well as *static*.

Index Mappings

A filed in document is equivalent to a column in a relational database table. All fields in a document needs to be mapped to a data type. ***Index Mapping*** define the data types of each field. This is equivalent to a table schema definition in a relational database. The *mapping* of an *index* can be declared explicitly or generated dynamically by elasticsearch using the index data.

Dynamic vs Explicit Mappings

Elasticsearch can auto generate the *mapping* for an *index* by looking at the data type of te ingested document fields. This is called a *dynamic mapping*. Once a field is mapped in a given index for a given data type that cannot be changed. Re-Indexing is the only option.

Therefore, *index mapping* can't be changed onced defined. Only option is to re-create the index and re-index documents.

> Tips Its always better to define *index mappings* explicitly. That helps to the for faster indexing of documents.

Let us now try to index some documents into our previously created my-index.

Indexing with a explicit document ID.

```
# Index a document with _id 1
PUT my-index/_doc/1
{
  "name": "Kasun",
  "age": 40,
  "city":"Melbourne",
  "country":"Australia",
  "weight": 80.6
}

## Index a document with _id 2
PUT my-index/_doc/2
{
  "name": "Amali",
  "age":  35,
  "city": "Sydney",
  "country": "Australia",
  "weight": 61.5
}

## Let's retrieve all the documents in the index
GET my-index/_search

## Also if we want we can index documents with auto generated IDs
POST my-index/_doc
{
  "name": "eranga",
  "age" : 41,
  "city": "Brisbane",
  "country": "Australia",
  "weight": 72.8
}
```

```
POST my-index/_doc
{
  "name": "Nimasha",
  "age" : 40,
  "city": "Canberra",
  "country": "Australia",
  "weight": 67.5
}

# Let's check the new values which got indexed.
GET my-index/_search
```

> Tips
>
> Its better to use autogenerated document IDs. That is more efficient and leads to better indexing performance. When explicit document IDs are used, elasticsearch needs to verify for the IDs to be unique. On the otherhand autogenerated IDs use time-based, universally unique identifiers (UUIDs), this removes the need for verficiation of uniqueness.

Problem with Dynamic Mappings

Earlier during our exercises we didn't specify a mapping explicitly. Therefore, elasticsearch generates a dynamic mapping based on the first document we indexed. Let's try to index the below document now.

```
POST my-index/_doc
{
  "name": "Yometh",
  "age": 6
  "city": "Perth",
  "country": "Australia",
  "weight": "18.5kg"
}
```

We should get an error. Let's see what was the data type which elasticsearch automatically mapped to the *age* field.

```
GET my-index/_mappings
```

We can see that the field *age* is mapped to type *long* and *weight* is mapped to type *float*. Therefore, value *"18kg" fails. Because when that value cannot be changed to a floating point value.

```
POST my-index/_doc
{
  "name": "Yometh",
  "age": 6
  "city": "Perth",
  "country": "Australia",
  "weight": "18.5"
}
```

But the above works. Even though its enclosed in double quotes, still since the value can be converted to a floating point value, elasticsearch accepts the document as a valid document.

Let's look at all the index documents again.

```
GET my-index/_search
```

We can see that elasticsearch keeps the original value of *weight* as a string with double quotes, even though it indexed the *weight* value as a floating point value.

Creating an index with Explicit Mappings

We can create an index with explicit mapping as below.

```
## We can create an index with explicit mappings.
PUT my-explicit-index
{
  "mappings": {
    "properties": {
      "year" : {
        "type" : "integer"
      },
      "city" : {
        "type" : "keyword"
      },
      "country" : {
        "type": "keyword"
      },
      "population_M" : {
        "type" : "keyword"
      },
      "attractions" : {
        "type" : "text"
      }
    }
  }
}
```

Let's see whether we can change the mapping once defined. Let's try to define the mapping again by changing the data type of population_M to a floating point.

```
PUT my-explicit-index
{
  "mappings": {
    "properties": {
      "year" : {
        "type" : "integer"
      },
      "city" : {
        "type" : "keyword"
      },
      "country" : {
        "type": "keyword"
      },
      "population_M" : {
        "type" : "float"
      },
      "attractions" : {
        "type" : "text"
      }
    }
  }
}
```

We should be getting an error, becase we are not allowed to change the mapped data type once defined. Only way out is to delete the index and re-create the mapping.

Data Types in Elasticsearch

- keyword

Keyword fields are string values. These will will be used for filtering documents, sorting and aggregations. Keyword fields will not be used for *full-text search*.

- text

Text fields are string values. These values will be *analyzed* and are used for *full-text search*.

- numeric

integer fields used to store numerical values. These will be mainly used for metric calculations, aggregations such as min, max and avg as well as range filters. Depending on the size of the numeric value, types such as integer, long, double and float can be used.

- date

elasticsearch accepts date fields in multiple formats. + a long value of milliseconds since the epoch + an integer value of seconds since the epoch + a formatted string value such as yyyy-MM-dd HH:mm:ss

- ip

Valid IPv4 and IPv6 values can be stored as ip fields in elasticsearch. IP ranges should be stored in as *ip_range* field in *CIDR Notation*

- boolean

True or False value.

- geo_point

Geo location data (lattitude, longititude pairs) can be mapped as geo_point on elasticsearch.

- object

This is used for complex data structures. *object* data type can be used to represent an inner object in the primary JSON document. Whenever a document field has a subfields will be mapped to type *object* by default. Objects allow you to clear.

example:

```
{
    "event": {
        "type": "http",
        "status": "complete"
    },
    "http": {
        "response": {
            "code": 500
        },
        "version": "1.1"
    },
    "@timestamp": "2023-10-23T01:12:23.431Z"
}

## Data is flattened inside elasticsearch as below

{
    "event.type": "http",
    "event.status": "complete",
    "http.response.code": 500,
    "http.version": "1.1",
    "@timestamp": "2023-10-23T01:12:23.431Z"
}
```

- array

When there is more than one value for each field, that can be stored in an array. Arrays do not need to be explicitly defined in the mapping. A field with any mapped data type can hold one or more values if required.

An array can only hold a singular data type. A field mapped as a string can only accept arrays with all string values.

example:

```
{
    "event": {
    "message": "CRM is up - 23 Oct 23",
    "status": "green"
    },
    "tags": ["Production", "Finance", "Nginx", "Web Server","US"]
}

## This will be flattened when indexed
{
    "event.message": "CRM is up - 23 Oct 23",
    "event.status": "green",
    "tags": ["Production", "Finance", "Nginx", "Web Server","US"]
}
```

- nested

The nested type allows you to index an array of objects, where the context of each object is preserved for querying. Standard query functions will not work in nested fields because of the internal representation of these fields. Therefore, queries that need to be run on nested fields should use the nested query syntax.

when indexing a document with nested fields, each object on a nested field is indexed as a separate document in an internal data structure. Due to this, if the document has a large array of nested objects, that will cause a large number of indexing operations.

Searching for nested object also can impact search performance. Therefore, denormalizing the data set could be looked at to improve the performance.

Let's see this in action now.

```
PUT stores
{
  "mappings": {
    "properties": {
      "store-name": {
        "type": "keyword"
      },
      "store-country":{
        "type": "keyword"
      },
      "product": {
        "type": "nested"
      }
    }
  }
}

# Let us now index a sample document into this index
POST stores/_doc
{
  "store-name": "Lawrance Foods",
  "store-country": "Sri Lanka",
  "product": [
    {
      "product": "chocolate gateau",
      "quantity": 21
    },
    {
      "product": "pineapple gateau",
      "quantity": 15
    }
  ]
}

POST stores/_doc
{
  "store-name": "Lawrance Foods",
  "store-country": "Singapore",
  "product": [
    {
      "product": "chocolate gateau",
      "quantity": 2
    },
    {
      "product": "coffee cake",
      "quantity": 15
    },
    {
      "product": "ribbon cake",
      "quantity": 18
    }
  ]
}

## Let's see the index documents
GET stores/_search



## Now let us try to query all stores that sell the chocolate gateau product
GET stores/_search
{
  "query": {
    "nested": {
      "path": "product",
      "query": {
        "bool": {
          "must": [
            {
              "match": {
                "product.product.keyword": "chocolate gateau"
              }
            }
          ]
        }
      }
    }
  }
}

## Even the below simple query works
GET stores/_search
{
  "query": {
    "nested": {
      "path": "product",
      "query": {
        "match": {
          "product.product.keyword": "coffee cake"
        }
      }
    }
  }
}
# searching for stores with coffee cakes

GET stores/_search
{
  "query": {
    "nested": {
      "path": "product",
      "query": {
        "bool": {
          "must": [
            {
              "match": {
                "product.product.keyword": "coffee cake"
              }
            }
          ]
        }
      }
    }
  }
}


# Let us try to query for store location Singapore
GET stores/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "store-country": "Singapore"
          }
        }
      ]
    }
  }
}

## Even below simple query works
## A bool query is not mandatory
GET stores/_search
{
  "query": {
    "match": {
      "store-country": "Singapore"
    }
  }
}


## Now let's return all the stores with less than 50 units of chocolate gateau
GET stores/_search
{
  "query": {
    "nested": {
      "path": "product",
      "query": {
        "bool": {
          "must": [
            {
              "match": {
                "product.product.keyword": "chocolate gateau"
              }
            },
            {
              "range": {
                "product.quantity": {
                  "lt": 50
                }
              }
            }

          ]
        }
      }
    }
  }
}

# Searching for count less than 10
GET stores/_search
{
  "query": {
    "nested": {
      "path": "product",
      "query": {
        "bool": {
          "must": [
            {
              "match": {
                "product.product.keyword": "chocolate gateau"
              }
            },
            {
              "range": {
                "product.quantity": {
                  "lt": 10
                }
              }
            }

          ]
        }
      }
    }
  }
}

# Let's try to get the inventory of the store in Sri Lanka
GET stores/_search
{
  "query": {
    "match": {
      "store-country": "Sri Lanka"
    }
  }
}

## Let's see what is the inventory level of i30 Sport
GET stores/_search
{
  "query": {
    "nested": {
      "path": "product",
      "query": {
        "match": {
          "product.product.keyword": "chocolate gateau"
        }
      }
    }
  }
}


## Let us now try to get a list of products where the inventory is less than 15
GET stores/_search
{
  "query": {
    "nested": {
      "path": "product",
      "query": {
        "bool": {
          "must": [
            {
              "range": {
                "product.quantity": {
                  "lt": 15
                }
              }
            }
          ]
        }
      }
    }
  }
}


## Now let's see whether we can simplify that query further
GET stores/_search
{
  "query": {
    "nested": {
      "path": "product",
      "query": {
        "range": {
          "product.quantity": {
            "lt": 15
          }
        }
      }
    }
  }
}
# It doesn't need to be a bool query
# bool query can be used if there are multiple conditions to be matched
```

- join

The join data enables us to create parent / child relationships across documents which we index. The condition is that all the related documents must exist on the shard within an index

Let's look at example, of a company data set. We have employees and their respective departments.

```
# Below is the document mapping
PUT department-employees
{
  "mappings": {
    "properties": {
      "dept_id": {
        "type" : "keyword"
      },
      "dept_name": {
        "type" : "keyword"
      },
      "employee_id" : {
        "type" : "keyword"
      },
      "employee_name" : {
        "type" : "keyword"
      },
      "employee_designation" : {
        "type" : "keyword"
      },
      "doc_type": {
        "type": "join",
        "relations": {
          "department": "employee"
        }
      }
    }
  }
}

## Let us now index some departments into the index

PUT department-employees/_doc/d1
{
  "dept_id": "D001",
  "dept_name": "Finance",
  "doc_type": "department"
}

PUT department-employees/_doc/d2
{
  "dept_id": "D002",
  "dept_name": "HR",
  "doc_type": "department"
}

PUT department-employees/_doc/d3
{
  "dept_id": "D003",
  "dept_name": "IT",
  "doc_type": "department"
}

# Let us now index some employees into the index

PUT department-employees/_doc/e1?routing=3
{
  "employee_id": "E001",
  "employee_name": "Kasun",
  "employee_designation": "Head Of IT",
  "doc_type": {
    "name": "employee",
    "parent": "d3"
  }
}

PUT department-employees/_doc/e2?routing=3
{
  "employee_id": "E002",
  "employee_name": "Amali",
  "employee_designation": "Manager Business Intelligence",
  "doc_type": {
    "name": "employee",
    "parent": "d3"
  }
}

PUT department-employees/_doc/e3?routing=2
{
  "employee_id": "E003",
  "employee_name": "Eranga",
  "employee_designation": "Head of HR",
  "doc_type": {
    "name": "employee",
    "parent": "d2"
  }
}

## We can use has_parent and has_child queries to run joined searches on your
data.

## To get a list of employees working for the IT department
GET department-employees/_search
{
  "query": {
    "has_parent": {
      "parent_type": "department",
      "query": {
        "term": {
          "dept_name": {
            "value": "IT"
          }
        }
      }
    }
  }
}

# To retrieve the department which Ben works for
GET department-employees/_search
{
  "query": {
    "has_child": {
      "type": "employee",
      "query": {
        "term": {
          "employee_name": {
            "value": "Eranga"
          }
        }
      }
    }
  }
}
```

Index Templates

An ***index template*** is a predifined schema and settings for an index which we can reuse by applying to a common set of indices.

This is commonly used for log indexing. *Index templates* can be automatically applied to new indices based on the name of the index.

Let's look at an example:

```
## Index Templates
## Creating an index template for all indices starting with the firewall-logs
string

PUT _index_template/logs-firewall
{
  "index_patterns": [
    "firewall-logs*"
  ],
  "template": {
    "settings": {
      "number_of_shards": 1
    },
    "mappings": {
      "properties": {
        "@timestamp": {
          "type": "date"
        },
        "source.ip": {
          "type": "ip"
        },
        "destination.ip": {
          "type": "ip"
        },
        "event.action": {
          "type": "keyword"
        },
        "user.name": {
          "type": "keyword"
        },
        "client.bytes": {
          "type": "double"
        }
      }
    }
  }
}


## Now let's create a new index by indexging a new document as below

POST firewall-logs-10.10.2023/_doc
{
 "@timestamp": "2023-10-23T01:26:26.231Z",
 "source.ip": "10.1.10.13",
 "destination.ip": "10.9.11.22",
 "event.action": "deny",
 "user.name": "kasun",
 "client.bytes": 5
}

## The new index got created and the record / document was indexed into it.
## let's now look at the index
GET firewall-logs-10.10.2023

## We can get the settings only as below
GET firewall-logs-10.10.2023/_settings

## We can get the mapping only as below
GET firewall-logs-10.10.2023/_mapping

## Let us now retrieve the record we indexed
GET firewall-logs-10.10.2023/_search
```

Elasticsearch Nodes

An ***elasticsearch node*** is a single running instance of elasticsearch. A single physical or virtual machine is capable of hosting either one or multiple instances of elasticsearch.

There are different types of node roles.

- Master eligible Nodes

At a given point, a single node is selected to be the *active master* from the master eligible node pool. The master node is responsible for + keeping track of other nodes in the cluster + creates or deletes indices + distributing shards based on requirements / constraints + maintaining cluster settings

As a best practice it is imporant to have more than one master eligible node to handle master node failures. The minimum number of master nodes required for a cluster is 3. If we have 2 then we have a problem which we call as *split brain* problem.

- Data Nodes

These are the nodes which keep the actual data / shards. The primary task of a *data node* is responding to read / write requests.

When we design enterprise grade *elasticsearch clusters* which include several *data nodes* we use *storage tiering* to categorize the *data nodes*. This is done, mainly to improve the performance as well as to keep the infrastructure cost down.

```
+ HOT Nodes (data hot)
    + continually written to
    + large volumes of data ingested per second
    + used for real-time use cases
    + generally use fast SSD / NVMe disks to improve throughput
+ WARM Nodes (data warm)
    + data doesn't get updated
    + used for data retention for long time periods with search / analytics
requirements
    + yet data can be interactivelly queried
    + uses higher density slower cheaper disks.
+ COLD Nodes (cold data)
    + utilize slower magnetic or network attached disks.
    + used to store infrequently accesssed data
    + data retained for longer retenso for compliance audit requirements
```

- Ingest Nodes

Used to process ***ingest pipelines*** associated with an indexing request. *Ingest pipelines* are used to transform incoming document data before getting indexed.

- Coordinator Nodes

Coordinator nodes routes saerch / indexing requests to the appropriate data node. Additionally they combine search results from multiple shards before returning the final result to the client.

- Machine Learning Nodes

Elasticsearch supports running machine learning jobs on top of real time data feeds. Machine learning nodes process the machine learning jobs and related API requests.

- Elasticsearch Clusters

A group of elasticsearch nodes put together is an elasticsearch cluster.

Searching for Data in Elasticsearch

Let us now look at a real world example of indexing some data and trying to search for data for analysis purpose.

Run the below script to get the data loaded into your elasticsearch cluster.

```
# Look at the current working directory
pwd
# Change your working directory to below path
cd /home/vagrant/web-logs
# Let's look at the execute permission of the script
ls -l
# We should have permissions as below
# -rwxrw-r-- 1 vagrant vagrant     210 Nov  7 04:17 ingest-web-log-data.sh
# -rwxrwxr-x 1 vagrant vagrant     781 Nov  6 12:22 setup-index-template-ingest-pipeline.sh
# If not run the below chmod command to add execution permission
# chmod u+x ingest-web-log-data.sh
# chmod u+x setup-index-template-ingest-pipeline.sh
#
# Then run the setup script first
# This will setup the required index and the mapping for the index
#
sudo ./setup-index-template-ingest-pipeline.sh
# Next let's run the script to load / index the web log data
sudo ./ingest-web-log-data.sh
```

Querying Data

Use Case 1 - Find all the HTTP events with an HTTP response code of 200 (Which is the status OK)

```
GET web-logs/_search
{
  "query": {
    "term": {
      "http.response.status_code": {
        "value": "200"
      }
    }
  }
}
```

Use Case 2 - Find all HTTP events where the request method was of POST type and resulted in a non - 200 response code (basically errors for POST requests)

```
GET web-logs/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "term": {
            "http.request.method": {
              "value": "POST"
            }
          }
        }
      ],
      "must_not": [
        {
          "term": {
            "http.response.status_code": {
              "value": "200"
            }
          }
        }
      ]
    }
  }
}
```

Use Case 3 - Find all HTTP events referencing the terms refrigerator and windows anywhere in the document

```
GET web-logs/_search
{
  "query": {
    "match": {
      "event.original": {
        "query": "refrigerator windows",
        "operator": "and"
      }
    }
  }
}
```

Use Case 4 - Look for all requests where users on Windows machines were looking at refrigerator related pages on the website.

```
GET web-logs/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "url.original.text": "refrigerator"
          }
        },
        {
          "match": {
            "user_agent.os.full.text": "windows"
          }
        }
      ]
    }
  }
}
```

Use Case 5 - Look for all events originating from either South Africa, Ireland or Hong Kong

```
GET web-logs/_search
{
  "query": {
    "terms": {
      "source.geo.country_name": [
        "South Africa",
        "Ireland",
        "Hong Kong"
      ]
    }
  }
}
```

Use Case 6 - Find all the events originating from IP addresses belonging to the Pars Online PJS and Respina Networks & Beyond PJSC telecommunication providers

```
## First we need to create a new index to keep the terms which we are going to
search

PUT telcos-list
{
  "mappings": {
    "properties": {
      "name": {
        "type": "keyword"
      }
    }
  }
}

## index a document containing the list of terms to be searched
PUT telcos-list/_doc/1
{
  "name": [
    "Pars Online PJS",
    "Respina Networks & Beyond PJSC"
  ]
}

GET web-logs/_search
{
  "query": {
    "terms": {
      "source.as.organization.name": {
        "index": "telcos-list",
        "id": "1",
        "path": "name"
      }
    }
  }
}
```

Use Case 7 - Find all HTTP GET events with response bodies of more than 100,000 bytes

```
GET web-logs/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "term": {
            "http.request.method": {
              "value": "GET"
            }
          }
        },
        {
          "range": {
            "http.response.body.bytes": {
              "gte": 10000
            }
          }
        }
      ]
    }
  }
}
```