Lab 3. Searching - What is Relevant

We will cover the following topics in this lab:

- Searching from structured data
- Writing compound queries
- Searching from full-text
- Modeling relationships

Standard tokenizer

The following example shows how the standard tokenizer breaks a character stream into tokens:

```
POST _analyze
{
  "tokenizer": "standard",
  "text": "Tokenizer breaks characters into tokens!"
}
```

The preceding command produces the following output; notice the `start_offset`, `end_offset`, and `positions` in the output:

```
{
  "tokens": [
    {
      "token": "Tokenizer",
      "start_offset": 0,
      "end_offset": 9,
      "type": "<ALPHANUM>",
      "position": 0
    },
    {
      "token": "breaks",
      "start_offset": 10,
      "end_offset": 16,
      "type": "<ALPHANUM>",
      "position": 1
    },
    {
      "token": "characters",
      "start_offset": 17,
      "end_offset": 27,
      "type": "<ALPHANUM>",
      "position": 2
    },
    {
      "token": "into",
      "start_offset": 28,
      "end_offset": 32,
      "type": "<ALPHANUM>",
      "position": 3
    },
    {
      "token": "tokens",
```

```
      "start_offset": 33,
      "end_offset": 39,
      "type": "<ALPHANUM>",
      "position": 4
    }
  ]
}
```

This token stream can be further processed by the token filters of the analyzer.

Standard analyzer

Let's see how Standard analyzer works by default with an example:

```
PUT index_standard_analyzer
{
  "settings": {
    "analysis": {
      "analyzer": {
        "std": {
          "type": "standard"
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "my_text": {
        "type": "text",
        "analyzer": "std"
      }
    }
  }
}
```

Here, we created an index, `index_standard_analyzer`.

Let's check how Elasticsearch will do the analysis for the `my_text` field whenever any document is indexed in this index. We can do this test using the `_analyze` API, as we saw earlier:

```
POST index_standard_analyzer/_analyze
{
  "field": "my_text",
  "text": "The Standard Analyzer works this way."
}
```

The output of this command shows the following tokens:

```
{
  "tokens": [
    {
      "token": "the",
      "start_offset": 0,
      "end_offset": 3,
```

```
      "type": "<ALPHANUM>",
      "position": 0
    },
    {
      "token": "standard",
      "start_offset": 4,
      "end_offset": 12,
      "type": "<ALPHANUM>",
      "position": 1
    },
    {
      "token": "analyzer",
      "start_offset": 13,
      "end_offset": 21,
      "type": "<ALPHANUM>",
      "position": 2
    },
    {
      "token": "works",
      "start_offset": 22,
      "end_offset": 27,
      "type": "<ALPHANUM>",
      "position": 3
    },
    {
      "token": "this",
      "start_offset": 28,
      "end_offset": 32,
      "type": "<ALPHANUM>",
      "position": 4
    },
    {
      "token": "way",
      "start_offset": 33,
      "end_offset": 36,
      "type": "<ALPHANUM>",
      "position": 5
    }
  ]
}
```

Please note that, in this case, the field level analyzer for the `my_field` field was set to Standard Analyzer explicitly. Even if it wasn't set explicitly for the field, Standard Analyzer is the default analyzer if no other analyzer is specified.

As you can see, all of the tokens in the output are lowercase. Even though the Standard Analyzer has a stop token filter, none of the tokens are filtered out. This is why the `_analyze` output has all words as tokens.

Let's create another index that uses English language stopwords:

```
PUT index_standard_analyzer_english_stopwords
{
  "settings": {
    "analysis": {
```

```
      "analyzer": {
        "std": {
          "type": "standard",
"stopwords": "_english_"
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "my_text": {
        "type": "text",
        "analyzer": "std"
      }
    }
  }
}
```

Notice the difference here. This new index is using *english* stopwords. You can also specify a list of stopwords directly, such as stopwords: (a, an, the). The *english* value includes all such English words.

When you try the _analyze API on the new index, you will see that it removes the stopwords, such as the and this:

```
POST index_standard_analyzer_english_stopwords/_analyze
{
  "field": "my_text",
  "text": "The Standard Analyzer works this way."
}
```

It returns a response like the following:

```
{
  "tokens": [
    {
      "token": "standard",
      "start_offset": 4,
      "end_offset": 12,
      "type": "<ALPHANUM>",
      "position": 1
    },
    {
      "token": "analyzer",
      "start_offset": 13,
      "end_offset": 21,
      "type": "<ALPHANUM>",
      "position": 2
    },
    {
      "token": "works",
      "start_offset": 22,
      "end_offset": 27,
      "type": "<ALPHANUM>",
      "position": 3
```

```
    },
    {
      "token": "way",
      "start_offset": 33,
      "end_offset": 36,
      "type": "<ALPHANUM>",
      "position": 5
    }
  ]
}
```

English stopwords such as `the` and `this` are removed. As you can see, with a little configuration, Standard Analyzer can be used for English and many other languages.

Let's go through a practical application of creating a custom analyzer.

Implementing autocomplete with a custom analyzer

If we were to use Standard Analyzer at indexing time, the following terms would be generated for the field with the `Course Elastic Search 7` value:

```
GET /_analyze
{
  "text": "Course Elastic Search 7",
  "analyzer": "standard"
}
```

The response of this request would contain the terms Course, Elastic, Search, and 7. These are the terms that Elasticsearch would create and store in the index if Standard Analyzer was used. Now, what we want to support is that when the user starts typing a few characters, we should be able to match possible matching products. For example, if the user has typed elas, it should still recommend Course Elastic Search 7 as a product. Let's compose an analyzer that can generate terms such as el, ela, elas, elast, elasti, elastic, le, lea, and so on:

```
PUT /custom_analyzer_index
{
  "settings": {
    "index": {
      "analysis": {
        "analyzer": {
          "custom_analyzer": {
            "type": "custom",
            "tokenizer": "standard",
            "filter": [
              "lowercase",
              "custom_edge_ngram"
            ]
          }
        },
        "filter": {
          "custom_edge_ngram": {
            "type": "edge_ngram",
            "min_gram": 2,
            "max_gram": 10
```

```
          }
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "product": {
        "type": "text",
        "analyzer": "custom_analyzer",
        "search_analyzer": "standard"
      }
    }
  }
}
```

This index definition creates a custom analyzer that uses Standard Tokenizer to create the tokens and uses two token filters -- a `lowercase` token filter and the `edge_ngram` token filter. The `edge_ngram` token filter breaks down each token into lengths of 2 characters, 3 characters, and 4 characters, up to 10 characters. One incoming token, such as elastic, will generate tokens such as [*el*], [*ela*], and so on, from one token. This will enable autocompletion searches.

Given that the following two products are indexed, and the user has typed `Ela` so far, the search should return both products:

```
POST /custom_analyzer_index/_doc
{
  "product": "Course Elastic Search 7"
}

POST /custom_analyzer_index/_doc
{
 "product": "Mastering Elasticsearch"
}

GET /custom_analyzer_index/_search
{
 "query": {
   "match": {
     "product": "Ela"
   }
 }
}
```

Before we move onto the next section and start looking at different query types, let's set up the necessary index with the data required for the next section. We are going to use product catalog data taken from the popular e-commerce site www.amazon.com.

Before we start with the queries, let's create the required index and import some data:

```
PUT /amazon_products
{
  "settings": {
```

```
    "number_of_shards": 1,
    "number_of_replicas": 0,
    "analysis": {
      "analyzer": {}
    }
  },
  "mappings": {
    "properties": {
      "id": {
        "type": "keyword"
      },
      "title": {
        "type": "text"
      },
      "description": {
        "type": "text"
      },
      "manufacturer": {
        "type": "text",
        "fields": {
          "raw": {
            "type": "keyword"
          }
        }
      },
      "price": {
        "type": "scaled_float",
        "scaling_factor": 100
      }
    }
  }
}
```

The `title` and `description` fields are analyzed text fields on which analysis should be performed. This will enable full-text queries on these fields. The `manufacturer` field is of the `text` type, but it also has a field with the name `raw`. The `manufacturer` field is stored in two ways, as `text`, and `manufacturer.raw` is stored as a `keyword`.

The `price` field is chosen to be of the `scaled_float` type. This is a new type introduced with Elastic 6.0, which internally stores floats as scaled whole numbers. For example, 13.99 will be stored as 1399 with a scaling factor of 100. This is space-efficient as `float` and `double` datatypes occupy much more space.

Import product data into Elasticsearch

1. Switch user from terminal: `su elasticsearch`
2. Logstash has been already downloaded at following path: `/elasticstack/logstash-7.12.1` and added to variable `PATH`.

https://www.elastic.co/downloads/logstash

3. Files have been already copied at path `/elasticstack/logstash-7.12.1/files`. The structure of files should look like -

```
/elasticstack/logstash-7.12.1/files/products.csv
/elasticstack/logstash-7.12.1/files/logstash_products.conf
```

4. Verify that index `amazon_products` is created by executing the command in the your Kibana - Dev Tools.

```
GET /amazon_products
```

6. Run logstash from command line, using the following commands:

```
cd /elasticstack/logstash-7.12.1

logstash -f files/logstash_products.conf
```

Verify Data Import

After you have imported the data, verify that it is imported with the following query:

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

In the next section, we will look at structured search queries.

Range query on numeric types

Suppose we are storing products with their prices in an Elasticsearch index and we want to get all products within a range. The following is the query to get products in the range of $10 to $20:

```
GET /amazon_products/_search
{
  "query": {
    "range": {
      "price": {
        "gte": 10,
        "lte": 20
      }
    }
  }
}
```

The response of this query looks like the following:

```
{
  "took" : 5,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
    "failed" : 0
```

```
  },
  "hits" : {
    "total" : {
      "value" : 201,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "amazon_products",
        "_type" : "_doc",
        "_id" : "q737OXkBtKf3dShzkvIb",
        "_score" : 1.0,
        "_source" : {
          "id" : "b0000dbykm",
          "title" : "mia's math adventure: just in time",
          "manufacturer" : "kutoka",
          "price" : "19.99",
          "description" : "in mia's math adventure: just in time children will help
mia save her house by using their math skills!"
        }
      },
...
```

Range query with score boosting

By default, the `range` query assigns a score of `1` to each matching document. What if you are using a `range` query in conjunction with some other query and you want to assign a higher score to the resulting document if it satisfies some criteria? We will look at compound queries such as the `bool` query, where you can combine multiple types of queries. The `range` query allows you to provide a `boost` parameter to enhance its score relative to other query/queries that it is combined with:

```
GET /amazon_products/_search
{
  "from": 0,
  "size": 10,
  "query": {
    "range": {
      "price": {
        "gte": 10,
        "lte": 20,
        "boost": 2.2
      }
    }
  }
}
```

All documents that pass the filter will have a score of `2.2` instead of 1 in this query.

Range query on dates

A `range` query can also be applied to date fields since dates are also inherently ordered. You can specify the date format while querying a date range:

```
GET /amazon_products/_search
{"query":{"range":{"orderDate":
{"gte":"01/09/2017","lte":"30/09/2017","format":"dd/MM/yyyy"}}}}
```

The preceding query will filter all the orders that were placed in the month of September 2017.

Note: We will get `"hits" : []` because no record is matched.

Elasticsearch allows us to use dates with or without the time in its queries. It also supports the use of special terms, including `now` to denote the current time. For example, the following query queries data from the last 7 days up until now, that is, data from exactly 24 x 7 hours ago till now with a precision of milliseconds:

```
GET /amazon_products/_search
{"query":{"range":{"orderDate":{"gte":"now-7d","lte":"now"}}}}
```

The ability to use terms such as `now` makes this easier to comprehend.

Exists query

Sometimes it is useful to obtain only records that have non-null and non-empty values in a certain field. For example, getting all products that have description fields defined:

```
GET /amazon_products/_search
{
  "query": {
    "exists": {
      "field": "description"
    }
  }
}
```

The `exists` query turns the query into a filter; in other words, it runs in a filter context. This is similar to the `range` query where the scores don't matter.

Term query

When we defined the `manufacturer` field, we stored it as both `text` and `keyword` fields. When doing an exact match, we have to use the field with the `keyword` type:

```
GET /amazon_products/_search
{
  "query": {
    "term": {
      "manufacturer.raw": "victory multimedia"
    }
  }
}
```

The `term` query is a low-level query in the sense that it doesn't perform any analysis on the term. Also, it directly runs against the inverted index constructed from the mentioned `term` field; in this case, against

the `manufacturer.raw` field. By default, the `term` query runs in the query context and hence calculates scores.

The response looks like the following (only the partial response is included):

```
{
  ...
  "hits": {
    "total" : {
      "value" : 3,
      "relation" : "eq"
    },
"max_score": 5.965414,
    "hits": [
      {
        "_index": "amazon_products",
        "_type": "products",
        "_id": "AV5rBfPNNI_2eZGciIHC",
        "_score": 5.965414,
   ...
```

As we can see, each document is scored by default. To run the `term` query in the filter context without scoring, it needs to be wrapped inside a `constant_score` filter:

```
GET /amazon_products/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "manufacturer.raw": "victory multimedia"
        }
      }
    }
  }
}
```

This query will now return results with a score of one for all matching documents. We will look at the `constant_score` query later in the lab. For now, you can imagine that it turns a scoring query into a non-scoring query. In all queries where we don't need to know how well a document fits the query, we can speed up the query by wrapping it inside `constant_score` with a `filter`. There are also other types of compound queries that can help in converting different types of queries and combining other queries; we will look at them when we examine compound queries.

Searching from the full text

We will cover the following full-text queries in the following sections:

- Match query
- Match phrase query
- Multi match query

Match query

When you use the `match` query on a `keyword` field, it knows that the underlying field is a `keyword` field, and hence, the search terms are not analyzed at the time of querying:

```
GET /amazon_products/_search
{
  "query": {
    "match": {
      "manufacturer.raw": "victory multimedia"
    }
  }
}
```

The `match` query, in this case, behaves just like a `term` query, which we understand from the previous section. It does not analyze the search term's `victory multimedia` as the separate terms `victory` and `multimedia`. This is because we are querying a `keyword` field, `manufacturer.raw`. In fact, in this particular case, the `match` query gets converted into a `term` query, such as the following:

```
GET /amazon_products/_search
{
  "query": {
    "term": {
      "manufacturer.raw": "victory multimedia"
    }
  }
}
```

The `term` query returns the same scores as the `match` query in this case, as they are both executed against a `keyword` field.

Let's see what happens if you execute a `match` query against a `text` field, which is a real use case for a full-text query:

```
GET /amazon_products/_search
{
  "query": {
    "match": {
      "manufacturer": "victory multimedia"
    }
  }
}
```

The `match` query with default parameters does all of these things to find the best matching documents in order, according to their scores (high to low).

By default, when only search terms are specified, this is how the `match` query behaves. It is possible to specify additional options for the `match` query. Let's look at some typical options that you would specify:

- Operator
- Minimum should match
- Fuzziness

Operator

The default behavior of the `match` query is to combine the results using the [*or*] operator, that is, one of the terms has to be present in the document's field.

This can be changed to use the `and` operator using the following query:

```
GET /amazon_products/_search
{
  "query": {
    "match": {
      "manufacturer": {
        "query": "victory multimedia",
        "operator": "and"
      }
    }
  }
}
```

In this case, both the terms `victory` and `multimedia` should be present in the document's `manufacturer` field.

Minimum should match

Instead of applying the `and` operator, we can keep the [*or*] operator and specify at least how many terms should match in a given document for it to be included in the result. This allows for finer-grained control:

```
GET /amazon_products/_search
{
  "query": {
    "match": {
      "manufacturer": {
        "query": "victory multimedia",
        "minimum_should_match": 2
      }
    }
  }
}
```

The preceding query behaves in a similar way to the `and` operator, as there are two terms in the query and we have specified that, as the minimum, two terms should match.

With `minimum_should_match`, we can specify something similar to at least three of the terms matching in the document.

Fuzziness

The following query has a misspelled word, `victor` instead of `victory`. Since we are using a `fuzziness` of `1`, it will still be able to find all `victory multimedia` records:

```
GET /amazon_products/_search
{
  "query": {
    "match": {
```

```
      "manufacturer": {
        "query": "victor multimedia",
        "fuzziness": 1
      }
    }
  }
}
```

If we wanted to still allow more room for errors to be correctable, the `fuzziness` should be increased to `2`. For example, a `fuzziness` of `2` will even match `victer`. `Victory` is two edits away from `victer`:

```
GET /amazon_products/_search
{
  "query": {
    "match": {
      "manufacturer": {
        "query": "victer multimedia",
        "fuzziness": 2
      }
    }
  }
}
```

The `AUTO` value means that the `fuzziness` numeric value of `0`, `1`, `2` is determined automatically based on the length of the original term. With `AUTO`, terms with up to 2 characters have `fuzziness` = `0` (must match exactly), terms from 3 to 5 characters have `fuzziness` = `1`, and terms with more than five characters have `fuzziness` = `2`.

Match phrase query

When you want to match a sequence of words, as opposed to separate terms in a document, the `match_phrase` query can be useful.

For example, the following text is present as part of the description for one of the products:

```
real video saltware aquarium on your desktop!
```

What we want are all the products that have this exact sequence of words right next to each other: `real video saltware aquarium`. We can use the `match_phrase` query to achieve it. The `match` query will not work, as it doesn't consider the sequence of terms and their proximity to each other. The `match` query can include all those documents that have any of the terms, even when they are out of order within the document:

```
GET /amazon_products/_search
{
  "query": {
    "match_phrase": {
      "description": {
        "query": "real video saltware aquarium"
      }
    }
  }
}
```

The response will look like the following:

```
{
  ...,
  "hits": {
    "total": 1,
    "max_score": 22.338196,
    "hits": [
      {
        "_index": "amazon_products",
        "_type": "products",
        "_id": "AV5rBfasNI_2eZGciIbg",
        "_score": 22.338196,
        "_source": {
          "price": "19.95",
          "description": "real video saltware aquarium on your desktop!product
information see real fish swimming on your desktop in full-motion video! you'll find
exotic saltwater fish such as sharks angelfish and more! enjoy the beauty and serenity
of a real aquarium at yourdeskt",
          "id": "b00004t2un",
          "title": "sales skills 2.0 ages 10+",
          "manufacturer": "victory multimedia",
          "tags": []
        }
      }
    ]
  }
}
```

The `match_phrase` query also supports the `slop` parameter, which allows you to specify an integer: 0, 1, 2, 3, and so on. `slop` relaxes the number of words/terms that can be skipped at the time of querying.

For example, a `slop` value of `1` would allow one missing word in the search text but would still match the document:

```
GET /amazon_products/_search
{
  "query": {
    "match_phrase": {
      "description": {
        "query": "real video aquarium",
        "slop": 1
      }
    }
  }
}
```

A `slop` value of `1` would allow the user to search with `real video aquarium` or `real saltware aquarium` and still match the document that contains the exact phrase `real video saltware aquarium`. The default value of `slop` is zero.

Querying multiple fields with defaults

The following query will find all of the documents that have the terms `monitor` or `aquarium` in the `title` or the `description` fields:

```
GET /amazon_products/_search
{
  "query": {
    "multi_match": {
      "query": "monitor aquarium",
      "fields": ["title", "description"]
    }
  }
}
```

This query gives equal importance to both fields. Let's look at how to boost one or more fields.

Boosting one or more fields

Let's make the `title` field three times more important than the `description` field. This can be done by using the following syntax:

```
GET /amazon_products/_search
{
  "query": {
    "multi_match": {
      "query": "monitor aquarium",
      "fields": ["title^3", "description"]
    }
  }
}
```

The `multi_match` query offers more control regarding how to combine the scores from different fields. Let's look at the options.

Writing compound queries

In this section, we will look at the following compound queries:

- Constant score query
- Bool query

Constant score query

For example, a `term` query is normally run in a query context. This means that when Elasticsearch executes a `term` query, it not only filters documents but also scores all of them:

```
GET /amazon_products/_search
{
  "query": {
    "term": {
      "manufacturer.raw": "victory multimedia"
    }
  }
}
```

Notice the text in bold. This part is the actual `term` query. By default, the `query` JSON element that contains the bold text defines a query context.

The response contains the score for every document. Please see the following partial response:

```
{
  ...,
  "hits": {
    "total": 3,
    "max_score": 5.966147,
    "hits": [
      {
        "_index": "amazon_products",
        "_type": "products",
        "_id": "AV5rBfasNI_2eZGciIbg",
        "_score": 5.966147,
        "_source": {
          "price": "19.95",
   ...
}
```

Here, we just intended to filter the documents, so there was no need to calculate the relevance score of each document.

The original query can be converted to run in a filter context using the following `constant_score` query:

```
GET /amazon_products/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "manufacturer.raw": "victory multimedia"
        }
      }
    }
  }
}
```

As you can see, we have wrapped the original highlighted `term` element and its child. It assigns a neutral score of `1` to each document by default. Please note the partial response in the following code:

```
{
  ...,
  "hits": {
    "total": 3,
"max_score": 1,
    "hits": [
      {
        "_index": "amazon_products",
        "_type": "products",
        "_id": "AV5rBfasNI_2eZGciIbg",
"_score": 1,
```

```
        "_source": {
          "price": "19.95",
          "description": ...
      }
   ...
}
```

It is possible to specify a `boost` parameter, which will assign that score instead of the neutral score of `1`:

```
GET /amazon_products/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "manufacturer.raw": "victory multimedia"

        }
      },
"boost": 1.2
    }
  }
}
```

Bool query

Let's first see how to implement simple `AND` and `OR` conjunctions.

A `bool` query has the following sections:

```
GET /amazon_products/_search
{
  "query": {
    "bool": {
      "must": [...],        scoring queries executed in query context
      "should": [...],      scoring queries executed in query context
      "filter": {},         non-scoring queries executed in filter context
      "must_not": [...]     non-scoring queries executed in filter context
    }
  }
}
```

The queries included in `must` and `should` clauses are executed in a query context unless the whole `bool` query is included inside a filter context.

Combining OR conditions

To find all of the products in the price range `10` to `13`, `OR` manufactured by `valuesoft`:

```
GET /amazon_products/_search
{
  "query": {
    "constant_score": {
```

```
      "filter": {
        "bool": {
          "should": [
            {
              "range": {
                "price": {
                  "gte": 10,
                  "lte": 13
                }
              }
            },
            {
              "term": {
                "manufacturer.raw": {
                  "value": "valuesoft"
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

Since we want to `OR` the conditions, we have placed them under `should`. Since we are not interested in the scores, we have wrapped our `bool` query inside a `constant_score` query.

Combining AND and OR conditions

Find all products in the price range `10` to `13`, `AND` manufactured by `valuesoft` or `pinnacle`:

```
GET /amazon_products/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "bool": {
          "must": [
            {
              "range": {
                "price": {
                  "gte": 10,
                  "lte": 30
                }
              }
            }
          ],
          "should": [
            {
              "term": {
                "manufacturer.raw": {
                  "value": "valuesoft"
```

```
                }
              }
            },
            {
              "term": {
                "manufacturer.raw": {
                  "value": "pinnacle"
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

Please note that all conditions that need to be ORed together are placed inside the `should` element. The conditions that need to be ANDed together, can be placed inside the `must` element, although it is also possible to put all the conditions to be ANDed in the `filter` element.

Adding NOT conditions

It is possible to add `NOT` conditions, that is, specifically filtering out certain clauses using the `must_not` clause in the `bool` filter. For example, find all of the products in the price range `10` to `20`, but they must not be manufactured by `encore`. The following query will do just that:

```
GET /amazon_products/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "bool": {
          "must": [
            {
              "range": {
                "price": {
                  "gte": 10,
                  "lte": 20
                }
              }
            }
          ],
          "must_not": [
            {
              "term": {
                "manufacturer.raw": "encore"
              }
            }
          ]
        }
      }
```

```
    }
  }
}
```

The `bool` query with the `must_not` element is useful for negate any query. To negate or apply a `NOT` filter to the query, it should be wrapped inside the `bool` with `must_not`, as follows:

```
GET /amazon_products/_search
{
  "query": {
    "bool": {
      "must_not": {
.... original query to be negated ...
      }
    }
  }
}
```

Notice that we do not need to wrap the query in a `constant_score` query when we are only using `must_not` to negate a query. The `must_not` query is always executed in a filter context.

Modeling relationships

In Elasticsearch, we can use the `join` datatype to model relationships. To import the data, follow the steps mentioned below:

Import product data into Elasticsearch

1. Switch user from terminal: `su elasticsearch`
2. Files have been already copied at path `/elasticstack/logstash-7.12.1/files` . The structure of files should look like -

```
/elasticstack/logstash-7.12.1/files/products_with_features_products.csv
/elasticstack/logstash-7.12.1/files/logstash_products_with_features_products.conf
/elasticstack/logstash-7.12.1/files/products_with_features_features.csv
/elasticstack/logstash-7.12.1/files/logstash_products_with_features_features.conf
```

The first two files are the files containing products data and logstash configuration file for loading products data respectively. The third and fourth files are the files containing data for features of the products and logstash configuration file for loading features data respectively.

4. Verify the `logstash_products_with_features_products.conf` file and ensure that it has the correct absolute path of products_with_features_products.csv file on your system. Similarly, verify the `logstash_products_with_features_features.conf` file and ensure that it has the correct absolute path of products_with_features_features.csv file on your system.
5. Create the following index by executing the command in the your Kibana - Dev Tools.

```
PUT /amazon_products_with_features
{
  "settings": {
    "number_of_shards": 1,
```

```
    "number_of_replicas": 0,
    "analysis": {
      "analyzer": {}
    }
  },
  "mappings": {
    "properties": {
      "id": {
        "type": "keyword"
      },
      "product_or_feature": {
        "type": "join",
        "relations": {
          "product": "feature"
        }
      },
      "title": {
        "type": "text"
      },
      "description": {
        "type": "text"
      },
      "manufacturer": {
        "type": "text",
        "fields": {
          "raw": {
            "type": "keyword"
          }
        }
      },
      "price": {
        "type": "scaled_float",
        "scaling_factor": 100
      },
      "feature_key": {
        "type": "keyword"
      },
      "feature": {
        "type": "keyword"
      },
      "feature_value": {
        "type": "keyword"
      }
    }
  }
}
```

6. Run the following commands from terminal:

```
cd /elasticstack/logstash-7.12.1
logstash -f files/logstash_products_with_features_products.conf
```

After running the second command press CTRL + C to terminate logstash. These commands would have loaded the products data into the Elasticsearch index. Next, let's load the features using the following command.

```
cd /elasticstack/logstash-7.12.1
logstash -f files/logstash_products_with_features_features.conf
```

We already populated products and features earlier in the lab, we can query from products while joining the data from features. For example, you may want to get all of the products that have a certain feature.

has_child query

Let's learn about this through an example. We want to get all of the products where `processor_series` is `Core i7` from the example dataset that we loaded:

```
GET amazon_products_with_features/_search
{
  "query": {
    "has_child": {
      "type": "feature",
      "query": {
        "bool": {
          "must": [
            {
              "term": {
                "feature_key": {
                  "value": "processor_series"
                }
              }
            },
            {
              "term": {
                "feature_value": {
                  "value": "Core i7"
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

The result of this `has_child` query is that we get back all the products that satisfy the query mentioned under the `has_child` element executed against all the features. The response should look like the following:

```
{
  "took" : 1,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "skipped" : 0,
```

```
    "failed" : 0
  },
  "hits" : {
    "total" : {
      "value" : 1,
      "relation" : "eq"
    },
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "amazon_products_with_features",
        "_type" : "doc",
        "_id" : "c0002",
        "_score" : 1.0,
        "_source" : {
          "description" : "The Acer G9-593-77WF Predator 15 Notebook comes with a 15.6
inch IPS display that makes each image and video appear sharp and crisp. The laptop
has Intel Core i7-6700HQ 2.60 GHz processor with NVIDIA Geforce GTX 1070 graphics and
16 GB DDR4 SDRAM that gives lag free experience. The laptop comes with 1 TB HDD along
with 256 GB SSD which makes all essential data and entertainment files handy.",
          "price" : "1899.99",
          "id" : "c0002",
          "title" : "Acer Predator 15 G9-593-77WF Notebook",
          "product_or_feature" : "product",
          "manufacturer" : "Acer"
        }
      }
    ]
  }
}
```

Next, let's look at another type of query that can be utilized when we have used the `join` type in Elasticsearch.

has_parent query

We want to get all the features of a specific product that has the product id = `c0003`. We can use a `has_parent` query as follows:

```
GET amazon_products_with_features/_search
{
  "query": {
    "has_parent": {
      "parent_type": "product",
      "query": {
        "ids": {
          "values": ["c0001"]
        }
      }
    }
  }
}
```

Please note the following points, marked with the numbers in the query:

1. Under the `has_parent` query, we need to specify the `parent_type` against which the query needs to be executed to get the children (features).
2. The `query` element can be any Elasticsearch query that will be run against the `parent_type`. Here we want features of a very specific product and we already know the Elasticsearch ID field (`_id`) of the product. This is why we use the `ids` query with just one value in the array: `c0001`.

parent_id query

In the `has_parent` query example, we already knew the ID of the parent document. There is actually a handier query to get all children documents if we know the ID of the parent document.

You guessed correctly; it is the `parent_id` query, where we obtain all children using the parent's ID:

```
GET /amazon_products_with_features/_search
{
  "query": {
    "parent_id": {
      "type": "feature",
      "id": "c0001"
    }
  }
}
```

Summary

In this lab, we took a deep dive into the search capabilities of Elasticsearch. We looked at the role of analyzers and the anatomy of an analyzer, saw how to use some of the built-in analyzers that come with Elasticsearch, and saw how to create custom analyzers. Along with a solid background on analyzers, we learned about two main types of queries --- term-level queries and full-text queries. We also learned how to compose different queries in more complex queries using one of the compound queries.