

26 MAY 2020

TECH TOPICS

# Improving search relevance with boolean queries

By **Alex Marquardt**

Share






When you perform a search in [Elasticsearch](#), results are ordered so that documents which are relevant to your query are ranked highly. However, results that may be considered relevant for one application may be considered less relevant for another application. Because Elasticsearch is super flexible, it can be fine-tuned to provide the most relevant search results for your specific use case(s). One relatively straightforward way to fine-tune results is by providing additional clauses in the queries that are sent to Elasticsearch.

In this blog, I'm going to walk you through brief examples to show you how easily you can improve search relevance using [boolean query](#) functionality in combination with [match queries](#) and [match phrase queries](#). If you'd like to follow along, but don't have your own cluster, you can spin up a [free trial of Elasticsearch Service](#) in a few minutes.

## Creating sample documents in Elasticsearch

To demonstrate the concepts in this blog, we first index several documents to Elasticsearch. These documents will be queried throughout this blog to demonstrate various concepts. Our demo documents can be written to Elasticsearch as follows:

### Recommended for you

-  [Elastic 7.15: Create powerful, personalized search experiences in seconds](#)
-  [Distributed Elastic Architectures](#)
-  [Twilio](#)
-  [Elasticsearch: Getting Started](#)
-  [Query string query | Elasticsearch](#)

### We're hiring

Work for a global, distributed team with impact? Development opportunities.

[Discover your career](#) →

```
PUT demo_idx/_doc/1
{
  "content": "Distributed nature, simple REST APIs, speed, and scalability"
}
PUT demo_idx/_doc/2
{
  "content": "Distributed nature, simple APIs, speed, and scalability"
}
PUT demo_idx/_doc/3
{
  "content": "Known for its simple REST APIs, distributed nature, speed, and
scalability, Elasticsearch is the central component of the Elastic Stack, a set of
open source tools for data ingestion, enrichment, storage, analysis, and
visualization."
}
```

Ok, now we have some data to work with. After completing this tutorial, you'll be able to apply these same techniques to your much larger data sets, but for now, we'll keep it simple.

## How documents are ranked in Elasticsearch

In order to understand the remainder of this blog, it is helpful to have a basic understanding of how Elasticsearch calculates a score that is used to order the documents returned by a query.

Before scoring documents, Elasticsearch first reduces the set of candidate documents by applying a boolean test that only includes documents that match the query. A score is then calculated for each document in this set, and this score determines how the documents are ordered. The score represents how relevant a given document is for a specific query. The default scoring algorithm used by Elasticsearch is [BM25](#). There are three main factors that determine a document's score:

1. **Term frequency (TF)** — The more times that a search term appears in the field we are searching in a document, the more relevant that document is.
2. **Inverse document frequency (IDF)** — The more documents that contain a search term in the field that we are searching, the less important that term is.
3. **Field length** — If a document contains a search term in a field that is very short (i.e. has few words), it is more likely relevant than a document that contains a search term in a field that is very long (i.e. has many words).

## A basic match query

A basic match query is often used for performing full-text search. By default, a match query with several terms will use an [OR operator](#) that will return documents that match *any* of the terms in the query. This may result in many documents being matched, even though some of the matched documents may only be slightly relevant. A search against the content field in the documents that we have just indexed would look similar to the following:

```
GET demo_idx/_search
{
  "query": {
    "match": {
      "content": {
        "query": "simple rest apis distributed nature"
      }
    }
  }
}
```

The above query will be interpreted as: **simple OR rest OR apis OR distributed OR nature**.

When we execute the above query the following results will be returned:

```
"hits" : {
  "total" : {
    "value" : 3,
    "relation" : "eq"
  },
  "max_score" : 1.2689934,
  "hits" : [
    {
      "_index" : "demo_idx",
      "_type" : "_doc",
      "_id" : "1",
      "_score" : 1.2689934,
      "_source" : {
        "content" : "Distributed nature, simple REST APIs, speed, and scalability"
      }
    },
    {
      "_index" : "demo_idx",
      "_type" : "_doc",
      "_id" : "2",
      "_score" : 0.6970792,
      "_source" : {
        "content" : "Distributed nature, simple APIs, speed, and scalability"
      }
    },
    {
      "_index" : "demo_idx",
      "_type" : "_doc",
      "_id" : "3",
      "_score" : 0.69611007,
      "_source" : {
        "content" : "Distributed nature, simple REST APIs, speed, and scalability"
      }
    }
  ]
}
```

[Read More](#) ↓

For many cases the above ordering may be exactly what is desired. In other cases, additional tuning may be required. The acceptability of different rankings will be dependent on the specific requirements of a given application.

The first hit is pretty good — it contains all of the words that we have searched for, although not in the order that we entered. The second hit is good, but notice that it is missing the word “rest” and the words are not the same order as what we searched for. And finally, the third hit could be considered a very good match for some use cases, as it contains all of the words that we have searched for in the exact order we typed them.

The reason that the third hit is not ranked higher than the first two hits is because of the following:

1. A match query using the OR operator does not take into account the position of the words. Therefore, even though the third hit (\_id: 3) contains the search text in the exact order that it was entered, this does not impact the score.
2. The third hit contains a relatively longer content field than the other hits. Therefore the *field length* portion of the scoring algorithm (which favours shorter fields) results in a lower score. In this example, the drop in the score for the third hit (\_id: 3) caused by its longer content field is larger than the drop in the score for the second hit (\_id: 2) caused by it missing the word “rest” in its content field.

Let’s see what happens if we use the AND operator in our match query.

## A match query that uses the AND operator

A search can be made more specific by using an [AND operator](#) in the match query. This will only return documents that contain *all* of the search terms. For a given query, the AND operator will return fewer documents than a match query that uses the OR operator. This means that the result set may miss some documents that the user may have considered relevant. An AND search against the content field in our index would look as follows:

```
GET demo_idx/_search
{
  "query": {
    "match": {
      "content": {
        "query": "simple rest apis distributed nature",
        "operator": "and"
      }
    }
  }
}
```

The above query will be interpreted as: *simple AND rest AND apis AND distributed AND nature*. When we execute the above query the following results will be returned:

```
"hits" : {
  "total" : {
    "value" : 2,
    "relation" : "eq"
  },
  "max_score" : 1.2689934,
  "hits" : [
    {
      "_index" : "demo_idx",
      "_type" : "_doc",
      "_id" : "1",
      "_score" : 1.2689934,
      "_source" : {
        "content" : "Distributed nature, simple REST APIs, speed, and scalability"
      }
    },
    {
      "_index" : "demo_idx",
      "_type" : "_doc",
      "_id" : "3",
      "_score" : 0.69611007,
      "_source" : {
        "content" : "Known for its simple REST APIs, distributed nature, speed, and
scalability, Elasticsearch is the central component of the Elastic Stack, a set of
open source
```

[Read More](#) ↓

This query has only returned two hits, and has excluded the second document that we ingested (`_id: 2`). This is because the second document does not contain the word “rest” in its content field, which is required for the AND condition to be satisfied. We now have more exact results, but we have removed a potentially relevant hit.

The second hit (`_id: 3`) could be considered more relevant than the first hit (`_id: 1`) as it contains the search terms in the exact order that they were entered. However, just like the OR operator, the AND operator does not consider the position of terms. Additionally, because the second hit has a relatively longer text field than the first hit, the *field length* portion of the scoring algorithm (which favours shorter fields) results in a lower score.

Let's see what happens if we use a match phrase query.

## The match phrase query

More exact results can be obtained by using the [match phrase query](#) which will only return documents that precisely match the phrase that a user is searching for. This is even more strict than a match query using the AND operator, and therefore will return fewer documents than either of the above queries. A match phrase query against document's content field would look similar to the following:

```
GET demo_idx/_search
{
  "query": {
    "match_phrase": {
      "content": {
        "query": "simple rest apis distributed nature"
      }
    }
  }
}
```

The above query will match documents that contain the phrase "simple rest apis distributed nature". In other words, only documents that contain all of the words in the same order as the search will be returned by the above query. Executing the above query returns the following result.

```
"hits" : {
  "total" : {
    "value" : 1,
    "relation" : "eq"
  },
  "max_score" : 0.6961101,
  "hits" : [
    {
      "_index" : "demo_idx",
      "_type" : "_doc",
      "_id" : "3",
      "_score" : 0.6961101,
      "_source" : {
        "content" : "Known for its simple REST APIs, distributed nature, speed, and scalability, Elasticsearch is the central component of the Elastic Stack, a set of open source tools for data ingestion, enrichment, storage, analysis, and visualization."
      }
    }
  ]
}
```

Notice that this query has only returned one hit. We now have a very specific result that matches exactly what the user was searching for, but this comes at the cost that other documents that may potentially be relevant are not returned.

It is possible that none of the above solutions give us the results that we are looking for. The remainder of this blog focuses on how to get more relevant search results by combining all of the above queries into a single query.

## Combining OR, AND, and match phrase queries

We may want exact matches to be ranked highly in our search results, but may also want to see documents that may be less relevant lower down in our results. Below we show how to use a should clause inside a [boolean query](#) to combine the OR, AND, and match phrase

queries to help us to meet our requirements. The should clause in a boolean query takes a more-matches-is-better approach, so the score from each clause will contribute to the final `_score` for each document.

The previous searches can be combined into a single should clause as follows:

```
GET demo_idx/_search
{
  "query": {
    "bool": {
      "should": [
        {
          "match": {
            "content": {
              "query": "simple rest apis distributed nature"
            }
          }
        },
        {
          "match": {
            "content": {
              "query": "simple rest apis distributed nature",
              "operator": "and"
            }
          }
        },
        {
          "match_phrase": {
            "content": {
              "query": "simple rest apis distributed nature"
            }
          }
        }
      ]
    }
  }
}
```

The above query evaluates each of the should clauses, and increases the score for each matching clause. Any document that is matched by the match phrase query will (by definition) also match the AND and the OR match queries. Likewise, any document that matches the AND will (by definition) also match the OR query. Therefore, we may anticipate that a document that matches the phrase that we have searched for will now be ranked higher than documents that do not match the phrase. However, the above query will return the following results, which may not be exactly as we expected:

```
"hits" : {
  "total" : {
    "value" : 3,
    "relation" : "eq"
  },
  "max_score" : 2.5379868,
  "hits" : [
    {
      "_index" : "demo_idx",
      "_type" : "_doc",
      "_id" : "1",
      "_score" : 2.5379868,
      "_source" : {
        "content" : "Distributed nature, simple REST APIs, speed, and scalability"
      }
    },
    {
      "_index" : "demo_idx",
      "_type" : "_doc",
      "_id" : "3",
      "_score" : 2.0883303,
      "_source" : {
        "content" : "Known for its simple REST APIs, distributed nature, speed, and scalability, Elasticsearch is the central component of the Elastic Stack, a set of open source
```

[Read More](#) ↓

This is pretty good, but we might not consider it to be perfect. We get hits for all of the relevant documents, but the hits are not ordered exactly as we had expected. We may have expected the second hit (`_id: 3`) to be ranked first. After all, the second hit exactly matches the phrase we have searched for (and therefore matches all of the `should` clauses), while the first hit (`_id: 1`) only matches the `AND` and the `OR` clauses. Why isn't the second hit (`_id: 3`) ranked first?

The documents are ranked in this order because the second hit (`_id: 3`) has a longer content field than the other hits, and therefore the score given to this document by each of the `should` clauses (`OR`, `AND`, and `match phrase`) has been proportionately reduced by the impact of the *field length* component of the scoring algorithm. In this case, the increased score due to a successful `match phrase` clause was not enough to offset this *field length* reduction in the score.

If we really want to ensure that exact matches are displayed before other hits, then we can boost individual clauses as described in the next section.

## Boosting individual clauses

A [boost](#) can be added to individual clauses to give them more importance. In our case, we wish to boost the `match phrase` clause, to ensure that documents that exactly match the phrase that we are searching for are returned first. This is accomplished with the following query:



```
GET demo_idx/_search
{
  "query": {
    "bool": {
      "should": [
        {
          "match": {
            "content": {
              "query": "simple rest apis distributed nature"
            }
          }
        },
        {
          "match": {
            "content": {
              "query": "simple rest apis distributed nature",
              "operator": "and"
            }
          }
        },
        {
          "match_phrase": {
            "content": {
              "query": "simple rest apis distributed nature",
              "boost": 2
            }
          }
        }
      ]
    }
  }
}
```

After executing the above query, we get results that look as follows:

```
"hits" : {
  "total" : {
    "value" : 3,
    "relation" : "eq"
  },
  "max_score" : 2.7844405,
  "hits" : [
    {
      "_index" : "demo_idx",
      "_type" : "_doc",
      "_id" : "3",
      "_score" : 2.7844405,
      "_source" : {
        "content" : "Known for its simple REST APIs, distributed nature, speed, and
scalability, Elasticsearch is the central component of the Elastic Stack, a set of
open source tools for data ingestion, enrichment, storage, analysis, and
visualization."
      }
    },
    {
      "_index" : "demo_idx",
      "_type" : "_doc",
      "_id" : "1",
      "_score" : 2.5379868,
      "_source"
```

[Read More](#) ↓

We have now received the results in the order that we are looking for. The document that contains the exact phrase that we searched for is the first hit. Additionally, we have received other less relevant documents lower down in the results.

## Using search templates

The above queries are getting to be rather large. Management of large or complex queries can be simplified by using [search templates](#). A search template for the above query would look as follows:

```
POST _scripts/demo_search_template
{
  "script": {
    "lang": "mustache",
    "source": {
      "query": {
        "bool": {
          "should": [
            {
              "match": {
                "content": {
                  "query": "{{query_string}}"
                }
              }
            },
            {
              "match": {
                "content": {
                  "query": "{{query_string}}",
                  "operator": "and"
                }
              }
            }
          ],
          "minimum_should_match": 2
        }
      }
    }
  }
}
```

The above search template can be executed with the following call:

```
GET _search/template
{
  "id": "demo_search_template",
  "params": {
    "query_string": "simple rest apis distributed nature"
  }
}
```

Which will return the exact same results as we previously received.

# View details of the score calculation

Elasticsearch provides an [explain API](#) and an [explain query parameter](#) to understand how the score is calculated. For example, explain can be executed with our basic match (OR) query as follows:

```
GET demo_idx/_search
{
  "explain": true,
  "query": {
    "match": {
      "content": {
        "query": "simple rest apis distributed nature"
      }
    }
  }
}
```

This will return a large and detailed response showing the various components of the score that are calculated for each matching document. However, analysis of the response is beyond the scope of this blog post.

## Other relevance tuning resources

For a more rigorous evaluation of the quality of the search results, the [Ranking Evaluation API](#) may be helpful. Additionally, more customized relevance scoring can be achieved as described in the [Easier Relevance Tuning in Elasticsearch 7.0](#) blog.

## Example project

A demonstration of the concepts presented in this blog can be found in the [ES Local Indexer project](#). This is a simple Python-based desktop search application that indexes html documents into Elasticsearch and that provides a browser-based interface for searching and paging through the ingested documents. Of particular relevance is the [search body](#) used in this project, which demonstrates many of the concepts discussed in this blog and that also demonstrates complex bool queries that search across multiple fields.

## Wrapping up (but you're just getting started)

By providing additional clauses in queries that are sent to Elasticsearch, it is possible to tune search results so that they are more relevant for a specific use case. In this blog I showed how to improve search relevance using basic [boolean query](#) functionality in combination with [match queries](#) and [match phrase queries](#). Start testing these methods out in your use cases today, and find out just how much better your users' search experience can be.

And if you prefer a pre-tuned search experience with drag and drop boosts and controls, check out [Elastic App Search](#). With App Search, you can implement highly relevant search experience with minimal effort, and it comes with an intuitive interface for tuning, curation, and analytics.

## Subscribe to our newsletter

Email address

Sign up

By submitting you agree to **Elastic Terms of Service**. Your personal data will be processed in accordance with **Elastic's Privacy Statement**.

### PRODUCTS & SOLUTIONS

- Enterprise Search
- Observability
- Security
- Elastic Stack
- Elasticsearch
- Kibana
- Logstash
- Beats
- Elastic Agent
- Subscriptions
- Pricing

### COMPANY

- Careers **WE'RE HIRING**
- Board of Directors
- Contact

### Follow Us



### RESOURCES

- ElasticON Global
- Documentation
- What is the ELK Stack?
- What is Elasticsearch?
- Migrating from Splunk
- Compare AWS Elasticsearch
- US Public Sector

### Language

English



Trademarks | Terms of Use | Privacy | Brand | Sitemap

© 2021. Elasticsearch B.V. All Rights Reserved

Elasticsearch is a trademark of Elasticsearch B.V., registered in the U.S. and in other countries.

Apache, Apache Lucene, Apache Hadoop, Hadoop, HDFS and the yellow elephant logo are trademarks of the **Apache Software Foundation** in the United States and/or other countries.