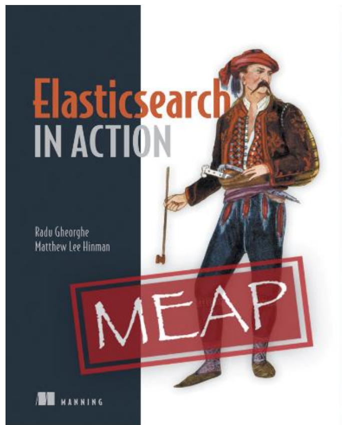
Solving search problems with Elasticsearch

By Radu Gheorghe, Roy Russo, and Matthew Lee Hinman, [Elastisearch in Action](#)



> As a developer, you know that search engines are confronted with challenges. In this article, we'll talk about Elasticsearch's approach to these challenges.

We use search everywhere now. Because of that, our expectations of search have grown. We want our searches to be smart and we want the results quickly. As a developer, you know that search engines are confronted with challenges. In this article, we'll talk about Elasticsearch's approach to these challenges.

To get a better idea of how Elasticsearch works, let's look at an example. Imagine that you're working on a website that hosts blogs, and you want to let users search across the entire site for specific posts. Your first task is to implement keyword search. For example, if a user searches for "elections," you'd better return all posts containing that word.

A search engine will do that for you, but for a *robust* search feature, you need more than that: results need to come in quickly, and they need to be relevant. And it's also nice to provide features that help users search when they don't know the exact words of what they're looking for. Those features include detecting typos, providing suggestions, and breaking down results into categories.

Providing quick searches

If you have a huge number of posts on your site, searching through all of them for the word "elections" can take a long time. And you don't want your users to wait. That's where Elasticsearch helps because it uses Lucene, a high-performance search engine library, to index all your data by default.

An index is a data structure, which you create along with your data and is meant to allow faster searches. You can add indices to fields in most databases, and there are several ways to do it. Lucene does it with *inverted indexing*, which means it creates a data structure where it keeps a list of where each word belongs. For example, if you need to search for blog posts by their tags, using inverted indexing might look like table 1.

Table 1 Inverted index for blog tags

Raw data		**Index data**	
Blog Post ID	Tags	Tags	Blog Post IDs
`1`	`elections`	`elections`	`1,3`
`2`	`peace`	`peace`	`2,3,4`
`3`	`elections, peace`		
`4`	`peace`		

If you search for blog posts that have an `elections` tag, it's much faster to look at the index rather than looking at each word of each blog post, because you only have to look at the place where tags is "elections," and you'll get all the corresponding blog posts. This speed gain makes sense in the context of a search engine. In the real world, you're rarely searching for one word only. For example, if you're searching for "Elasticsearch in Action," three-word look ups imply multiplying your speed gain by three.

An inverted index is appropriate for a search engine when it comes to relevance, too. For example, when you're looking up a word like "peace," not only will you see which document matches, but you'll also get the number of matching documents for free. This is important because if a word occurs in most documents, it's probably less relevant. Let's say you search for "Elasticsearch in Action," and a document contains the word "in"—along with a million other documents. At this point, you know that "in" is a common word, and the fact that this document matched doesn't say much about how relevant it is to your search. In contrast, if it contains "Elasticsearch," along with a hundred others, you know you're getting closer to relevant documents. Although, it's not "you" who has to know you're getting closer; Elasticsearch does that for you.

That said, the tradeoff for improved search performance and relevancy is that the index will take up disk space, and adding new blog posts will be slower because you have to update the index after adding the data itself. On the upside, tuning can make Elasticsearch faster, both when it comes to indexing and searching.

Ensuring relevant results

Then there's the hard part: How do you make the blog posts that are about elections appear before the ones that merely contain that word? With Elasticsearch, you have a few algorithms for calculating the *relevancy score*, which is used, by default, to sort the results.

The relevancy score is a number assigned to each document that matches your search criteria and indicates how relevant the given document is to the criteria. For example, if a blog post contains "elections" more times than another, it's more likely to be about elections.

By default, the algorithm used to calculate a document's relevancy score is *tf-idf*. Here's the basic idea: *tf-idf* stands for *t*erm *f*requency–*i*nverse *d*ocument *f*requency, which are the two factors that influence relevancy score:

- *Term frequency*—The more times the words you're looking for appear in a document, the higher the score.
- *Inverse document frequency*—The weight of each word is higher if the word is uncommon across other documents.

For example, if you're looking for "bicycle race" on a cyclist's blog, the word "bicycle" counts much less for the score than "race." But the more times both words appear in a document, the higher that document's score.

In addition to choosing an algorithm, Elasticsearch provides many other built-in features to influence the relevancy score to suit your needs. For example, you can "boost" the score of a particular field, such as the title of a post, to be more important than the body. This gives higher scores to documents that match your search criteria in the title, compared to similar documents that match only the body. You can make exact matches count more than partial matches, and you can even use a script to add custom criteria to the way the score is calculated. For example, if you let users like posts, you can boost the score based on the number of likes, or you can make newer posts have higher scores than similar, older posts.

Searching beyond exact matches

Finally, with Elasticsearch you have options to make your searches intuitive and go beyond exactly matching what the user types in. These options are handy when the user enters a typo or uses a synonym or a derived word different than what you've stored. And they're also handy when the user doesn't know exactly what to search for in the first place.

HANDLING TYPOS

You can configure Elasticsearch to be tolerant of variations instead of looking for only exact matches. A fuzzy query can be used so a search for "bicycel" will match a blog post about bicycles.

SUPPORTING DERIVATIVES

You can also use analysis to make Elasticsearch understand that a blog with "bicycle" in its title should also match queries that mention "bicyclist" or "cycling."

USING STATISTICS

When users don't know what to search for, you can help them in a number of ways. One way is to present statistics through aggregations. Aggregations are a way to get counters from the results of your query, like how many topics fall into each category or the average number of likes and shares for each of those categories. Imagine that upon entering your blog, users see popular topics listed on the right-hand side. One topic may be cycling. Those interested in cycling would click that topic to narrow the results. Then, you might have another aggregation to separate cycling posts into "bicycle reviews," "cycling events," and so on.

PROVIDING SUGGESTIONS

Once users start typing, you can help them discover popular searches and popular results. You can use suggestions to predict their searches as they type, like most search engines on the web do. You can also show popular results as they type, using special query types that match prefixes, wild cards, or regular expressions.