Jedan od prvih Search Engine-a je program koji se zove Archi koji je objavljen 1990. i omogucavao je pristup i pretrazivanje imena fajlova koji su tehnicki imena web page-ova. Archi nije mogao da kaze sta se tacno nalazi tim page-ovima.

Danasnji Search Engine-i: Google, Bing, Yahoo ...

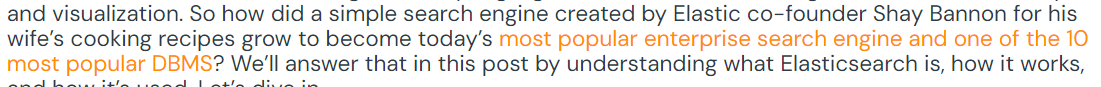
Search query - otkucana rec/fraza

Indexing - a process where search engine organizes and stores online content in a central db(its index). Search engine can then analyze and understand content and server it to the readers in ranked lists on its Search engine results pages(SERPs).

When web page is being indexed it means it's been crawled and analyzed by web crawlers like Googlebot or Bingbot and then added to its database of search results. Site pages have to be indexed before before they can populate in a search engine's results page. Owner of a website can request a page to be indexed or a page can be discovered by a search engine bot through inbound or internal linking.

Search engine features:

* **Full-text:**can be a search by a single word, multiple word forms, or a sentence.
* **Fuzzy search:** a technique that helps to return relevant search results for queries containing mistakes and misprints.
* **Geospatial search:** automatically detects the user's geolocation.
* **Faceted search:** allows to expand the search with such attributes as size, color, and others (search by brand name, vendor, product rating, etc.)
* **Sorting feature:** consists of a search by range (for example, by a price range, size range or time interval), and a filter-assisted search (to include only the desirable parameters).
* **Near real-time (NRT) indexing:** considers the frequent updates (such as constantly changing information, for example, availability in stock, dynamically changing prices, or product description details).
* **Highlighting:** visual indication of the words entered in the search bar, snippets.
* **Security:** ability to identify threats like web server problems or broken links, and show them on a dashboard.
* **Scalability:**opportunity to scale the database over time without hampering the search speed.
* **Storage:** database search is perceived as an additional source to store data.



<https://searchanise.io/blog/3-most-popular-search-engine-technologies-in-2020/>

**ElasticSearch**, **Splunk**, **Solr**, **MarkLogic**, **Algolia**, **Microsoft Azure Search**, **Sphinx**, **Arango**, **Virtuoso**, **OpenSearch**

**ElasticSearch** - open source, analythics engine that is mostly used, rapidly indexes changing data in < 1s = fastes search engine, good to use when db is updating constantly, scalable = search speed suffers with db growth. Stack Overflow (they chose it because of perceived performance and the ability to customize search paramters), Uber and Expedia use this search engnie. Can be used in .NET, Java, Pythong, SQL and PHP.

**Splunk** - indexes and searches log files stored in a system. Monitors, analyzes and searches machine-generated BigData in real-time using web-style interface. Provides instant results, allows generating alerts, graphs, dashboards, produces metrics, detects problem to get actionable data. It's easy to incorporate AI into the data strategy. Not user-friendly, free documentation is not enough, paid training is needed too, debugging/trouble shooting is hard.

**Solr** - stand-alone open source from Apache Lucene project, supports rich schema specification, has extensive search plugin API for developing custom search behaviour. Provides stopwords, synonyms and quality faceted search. Not as fast as ElasticSearch and its best to be used for databases with static data, which means no frequent changes of data in db. Reddit, Netflix, Zappos and Macy's use this search engine. Integrates well with Apache Hadoop and NoSQL database.

**Sphinx** - open source, full-text search, high search relevancy, rapid speed (500 queries/sec), simplicity of integration, support of various query types (phrase or word proximity), faceted search. It comes with editable tables, directives for building a hierarchy of data, dynamic linking for cross-referencing, extensive support for setting up the software. Good for web sites that use a lot of documentation. Lacks data visualization and analytics tools.

**MarkLogic** - multi-model enterprise db technology coupled with search and application services. Besides storing, provides built-in search and application services. Distributed architecture gives ability to process huge amount of data. Government, financial and publishing sectors use this search engine. Best usage is for storing and searching through enormous amount of data.

**Algolia** - hosted software as a Service search. Easy to implement, typo correction, search analytics, facets. ElasticSearch is often compared to this one, but some users more prefer it because of more affordable pricing and greater flexibility.

**Microsoft Azure Cognitive Search** - a Service framework, provides indexing and querying capabilites for data that's being uploaded on Microsoft servers. Includes features such as suggestions, facets, geosearch, synonyms etc. It can handle modern query syntaxes and natural language queries. Highly customizable and flexible solutio with advanced data security. eBay, Samsung, Pixar and Apple iCloud use this search engine. Best usage is for enterprises that are sticking with Microsoft ecosystem.

(the rest of them are less used than the previous ones)

**ArangoSearch** - full-text search integrated into ArangoDB. Two information retrival techniques: boolean and generalizd ranking retrival. Results can be ranked by relevance.

**Virtuoso** - hybrid RDMS that specializes in AI-driven data visualization, which means it helps to create gaphical representation of information and data. Users can manage data represented as relational tables and property graphs. 'Snappy speed is a definite plus'.

**OpenSearch** - search and analytics engine that split from ElasticSearch.

<https://www.youtube.com/@msci541-searchengines3/videos>

Internal search ULRs should not be indexed. Reasons for it are use experience and , as John Muller, webmaster at Google, said, Google's limited crawl budget for a website can be easily eaten by enormous number of internal search ULRs.

<https://stackoverflow.blog/2008/09/21/what-was-stack-overflow-built-with/>

Stack Overflow

* ElasticSearch
* C# - ASP.NET MVC framework
* SQL Server database
* the Dapper ORM for data access

IMENA ZA PROJEKAT - CodeMate, Stack Net, InfoLoop, CodeAid, DebugIt, Debug Station

benefits issues

Tip za id polje - GUID - fixed length, global uniqueness; low search performance

(for migrating data to another location)

Int - fast for select/update, minimal hard disk space

**MIGRACIJE**

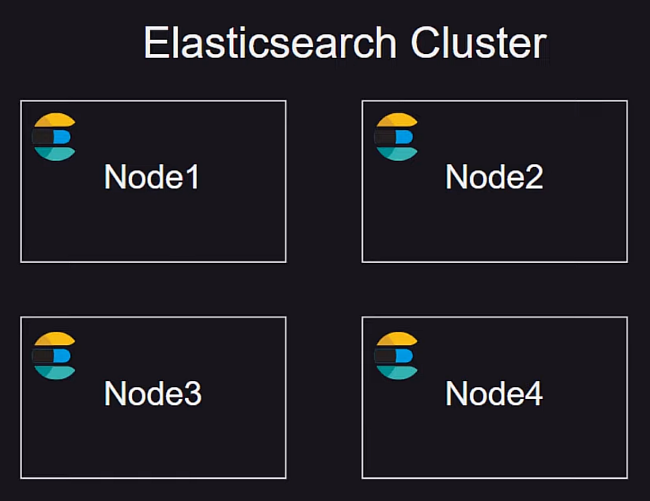
Add-Migration InitialMigration -c DebugItContext -o Migrations

Update-Database

**Elasticsearch engine**

Elasticsearch is a distributed database where data is stored as JSON documents.

Distributed database means that database can run at multiple nodes(computing devices/servers that are part of the database system) at the time.



**Cluster** = 1 or more Elasticsearch nodes connected together.

4 Nodes = 4 servers. **Node** = server running in Elasticsearch instance. You can have replication or add more nodes.

Elasticsearch distributes it's workload across multiple nodes in a cluster, which means it's horizontally scallable.

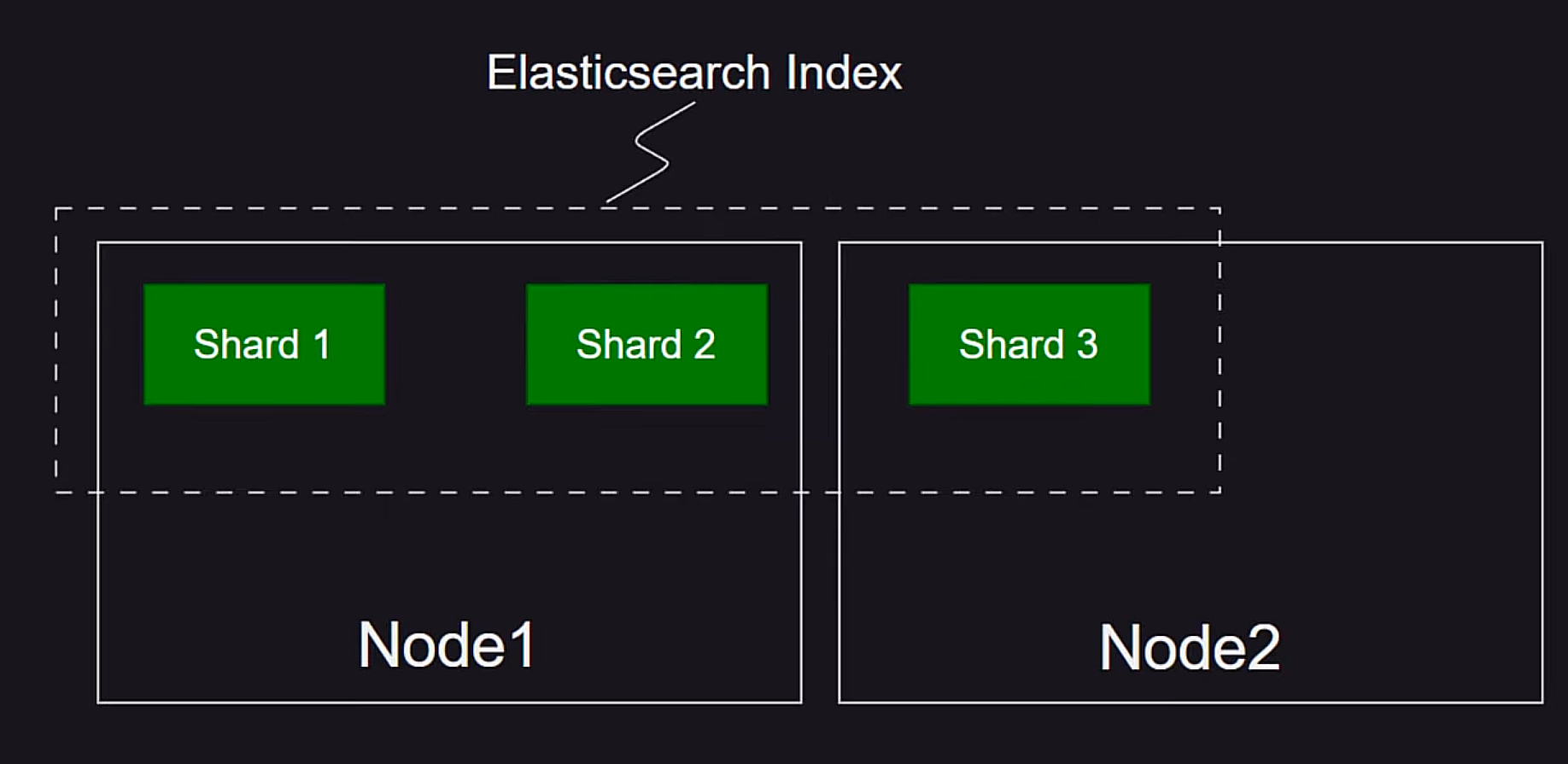
Fundemental unit of data - JSON documents (it's like a row in a db table).

**Index** - something like db table, a group of documents in an Elasticsearch db, an optimized collection of documents where each document is a collection of fileds which are represented as a key-value pairs.



Data in an Elasticsearch index can grow to massive proportions and in order to kep it managable, the data is split into a number of shards.

**Shard** - fundamental building block of Elasticsearch's distributed architecture. One Index can have 1 or more shards.



Shards are gonna be balanced between the nodes. For one query, only one node with it's shards needs to be responsible for securying result, which means that having mulitple nodes and multiple shards means that there can be parallel queries. Shards and indexes can be configurable and there can be only one shard per index.

[About shards](https://opster.com/guides/elasticsearch/glossary/what-are-shards-in-elasticsearch/) - check for Best Practices

Types of shards:

- primary - hold the original data and are responsible for indexing and searching operations. Number of primary shards is determined at the time of index creation and cannot be changed afterward.

- replicas - readonly shards used for parallel queries. They are the copies of primary shards, created to provide redundancy anad improve search performance. They can be added or removed dinamically and Elasticsearch automatically balances them across the nodes in the cluster. There should be at least one replica shard for each primary shard.

[More details about shards](https://opster.com/guides/elasticsearch/glossary/elasticsearch-shards/)

The ideal number of shards should be determined based on the amount of data in an index. Generally, an optimal shard should hold 30-50GB of data.

<https://www.elastic.co/guide/en/elasticsearch/reference/7.17/size-your-shards.html#:~:text=The%20number%20of%20shards%20a,keep%20your%20nodes%2C%20the%20better.>

The number of shards a data node can hold is proportional to the node's heap memory. A node with 30GB of heap memory should have at most 600 shards. The further below this limit a node can be kept, the better. If node exceeds more than 20shards per GB, then it's supposed to be considered adding another node.

When queries are run across different shards in parallel, they execute faster than an index compoed of a single shard, but only if each shard is located on a different node and there are sufficient nodes in the cluster.

Shards consume memory and disk space, both in terms of indexed data and cluster metadata(describes how to read the data stored on the data nodes, if lost, data stored on the data nodes cannot be read). Having too many shards can slow down queries, indexing requests and management operation. That's why the right balance of shards is very important.

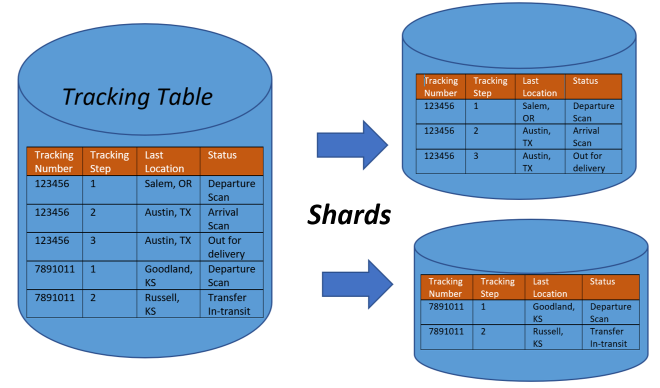
Benefits of Sharding:

- horizontal scalling - shards allow Elasticsearch to distribute data across mulitple nodes, enabling the cluster to grow horizontally as the data volume increases. This ensures that no single node becomes a bottleneck, thus maintaining optimal performance.

- improved search performance - by splitting the data into smaller units, Elasticsearch can execute search queries on multiple shards concurently, resulting in faster response times.

- high availability - replica shards ensure that the data remains available even if a node fails, which can be due to issues like insufficient resources, network connectivity problems or configuration errors. Elasticsearch automatically routes search and indexing operations to the available shards, maintaining the cluster's overall health.

Sharding is just another name for 'horizontal partitioning'. It's a method of splitting and storing a single logical dataset in multipl databases.



Horizontal pratitioning - design principle whereby rows of a database table are held sparately. Each partition forms a part of a shard, which may be located on a separate database server of physcal location. ([Wikipedia](https://en.wikipedia.org/wiki/Shard_(database_architecture)))

**Why use Elasticsearch db?**

- It's a distributed db and it's horizontally scalable, which is not possible with relational dbs.

- It's designed for faster queries.

Elasticsearch stores data in data structure called inverted index, where data is literally stored as searches. This makes querying very fast even if vast amounts of data storage.

Example 1

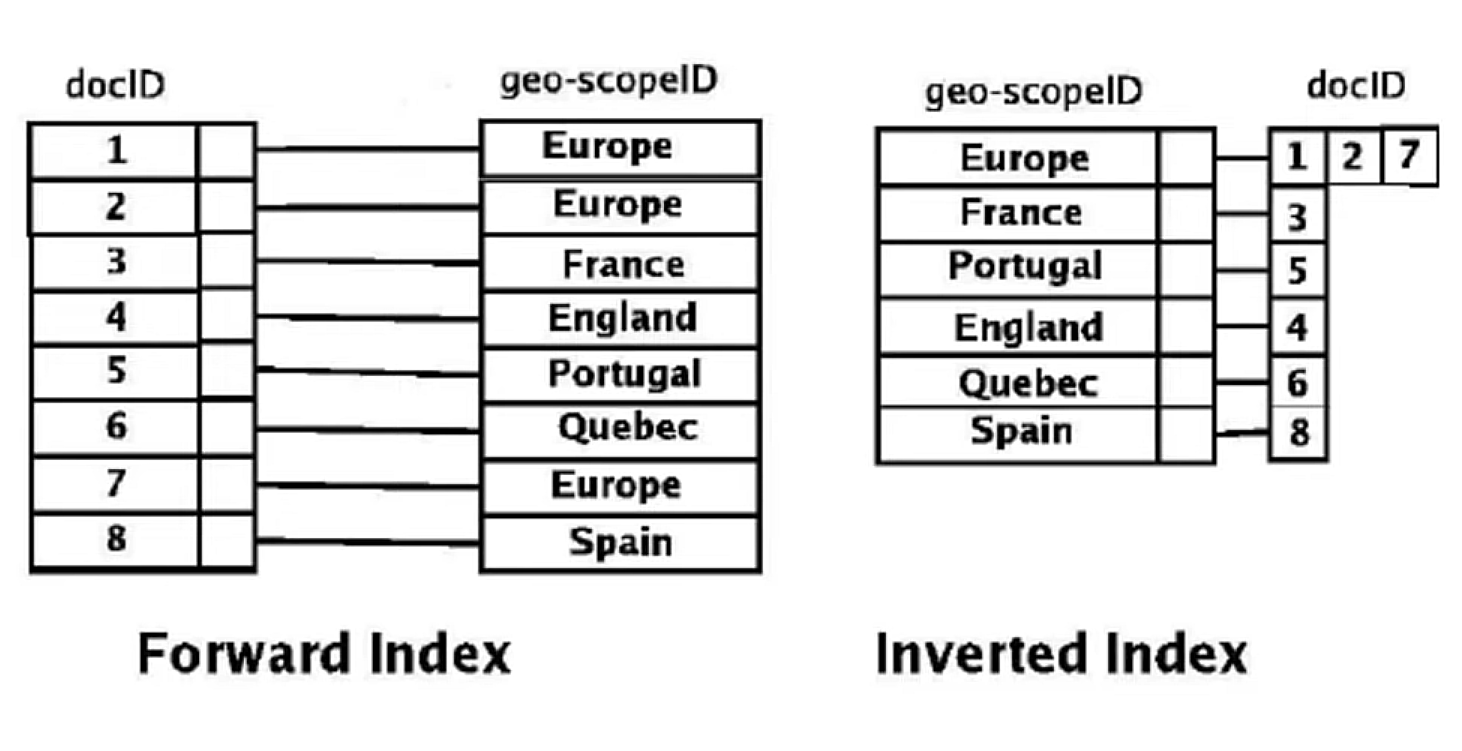
There are 2 data structures/methods for storing and organizing data:

- forward index - documents are mapped to the terms they contain. For example, in search engine, this method stores a list of all the words contained in each document. It's good to use it when documents need to be retreived or have their content analyzed. They're not very efficient for searching by term.

- inverted index - terms are mapped to the documents that contain them. Structure like this is crucial for fast search operation, like in search engines. It stores a list of documents associated with each term.

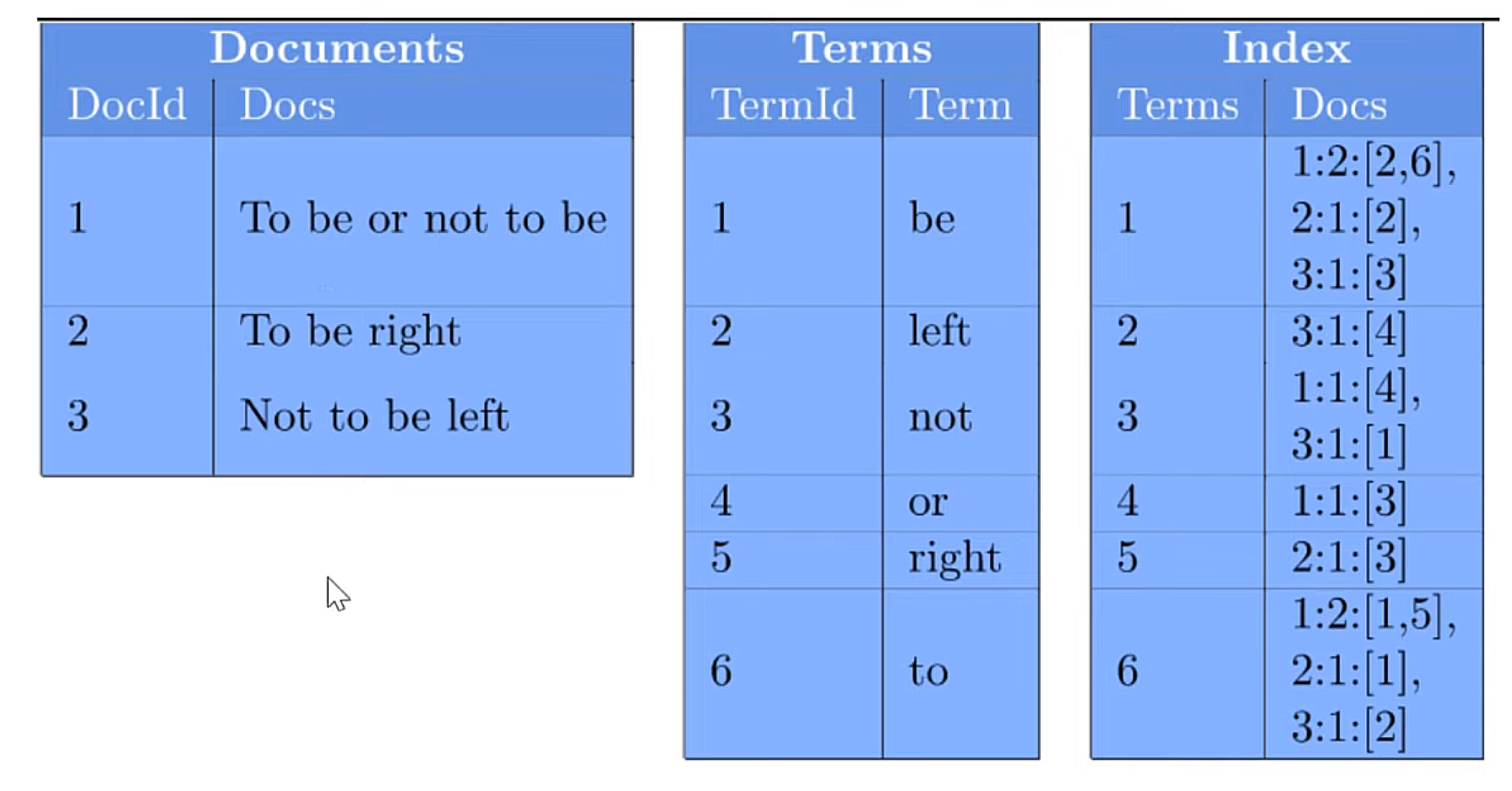
There are 8 documents with attribute called geo-scopeID. If data is stored using the forward index method and if a user wants to search for all the documents that contain the word Europe, he would have to go through all the documents just to get values 1,2 and 7.

In Elasticsearch, data is already stored as search(inverted index method) and when a new query comes, it can instantly be seen that the word Europe is in documents 1, 2 and 7. So, inverted index is a way that allows fast full-text searches at a cost of increased processing when a document is added to the db.



Example 2

There are 3 documents with some words in them. What Elasticsearch does is it tokenizes them and that means it finds unique words. Then it stores those words like it's shown in Index table, where first column represents the document, second column represents the frequency and the third column represents location of a word. Storing data like this is like storing the search, which is why searching with elasticsearch faster.(more about it on the end of page 10)



Data types:

- core types - text, keyord, date, long, double and all that represents the fundamental data types

- complex types - object, nested (specialized version of object data that allows arrays of objects to be indexed in a way that they can b queried independently of each other) - allow for structured JSON data to be stored

- specialized types

Lucene/Apache Lucen is an open-source Java library used as a search engine. Elasticsearch is built on top of Lucene. It converts Lucene into distributed system/search engine for scaling horizontally.

<https://stackoverflow.com/questions/56619574/do-all-shards-within-index-have-the-same-content#:~:text=Do%20all%20shards%20(within%20index)%20have%20the%20same%20content.,)%2C%20let%27s%20take%20an%20example.>

Example 3

Primary shards of one index do not have the same content/data in them.

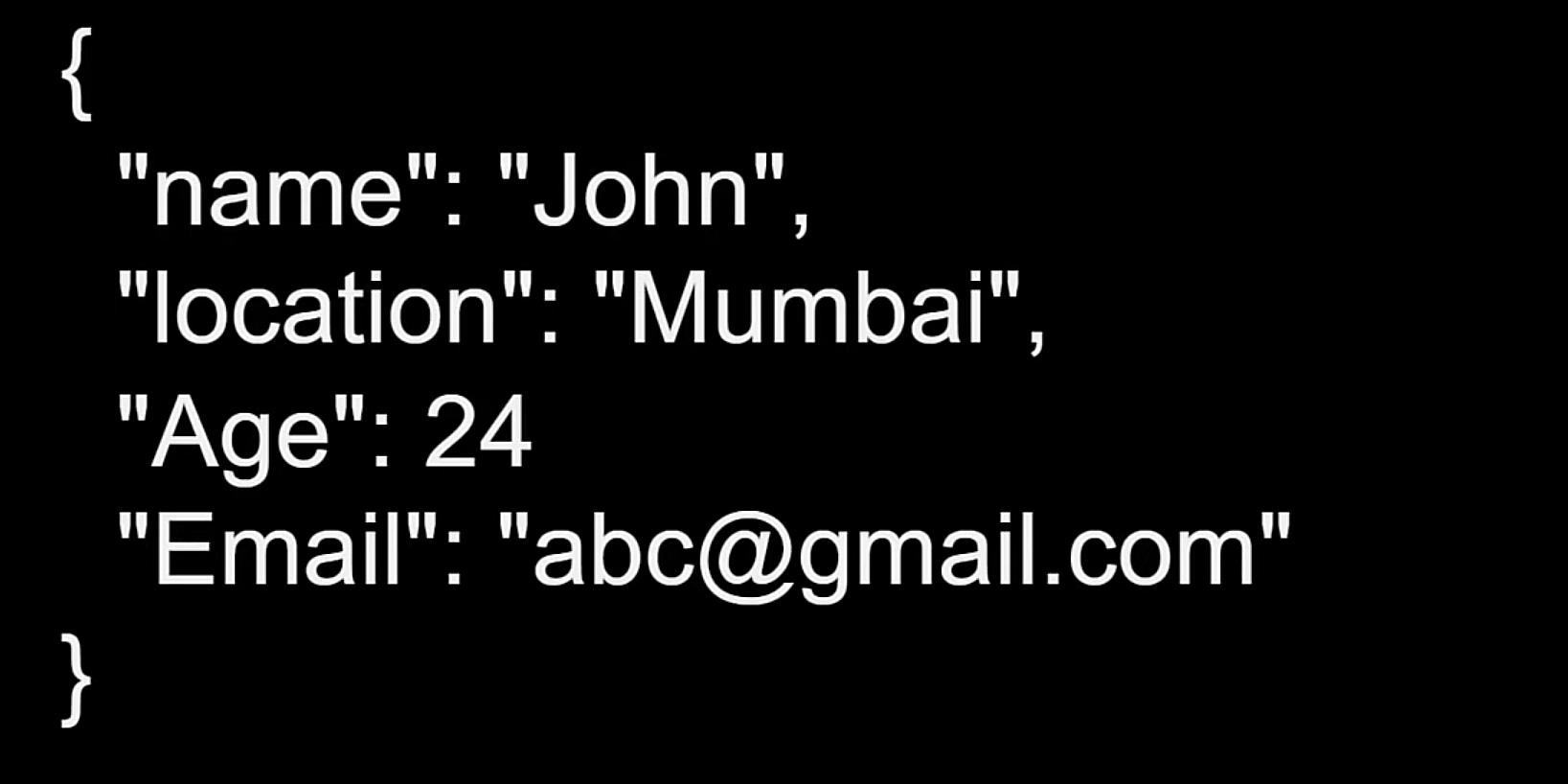
If cluster has 10 nodes and 100 milion docs, there can be, for example, 10 primary shards and for each primary, there should be a replica, so there should be also 10 replicas. Elasticsearch will physically divide data into those 10 shards. Since there are 10 nodes and 10 primary shard, each node will have 1 primary shard, but each replica of that primary shard will be on a different node to the one it's primary shard is on.

**Index template** contains settings like number of shards and replicas, data mapping, priority etc. that are applied to the index while creating.

**Index data mapping** is used to specify the schema of an index. It can be set to be dynamic, which means that as documents are arriving to the db, Elasticsearch is trying to realize what schema should be and it's generating/updating the schema. This is also called Schema on Write. It can be also set to strict(explanation in Example 4).

Example 4

Index data mapping is set to dynamic. This is the document that arrived, so Elasticsearch can create a schema from it where it says that there should be fields 'name' with a type string, 'location' with a type string, 'Age' with a type number and 'Email' with a type string.



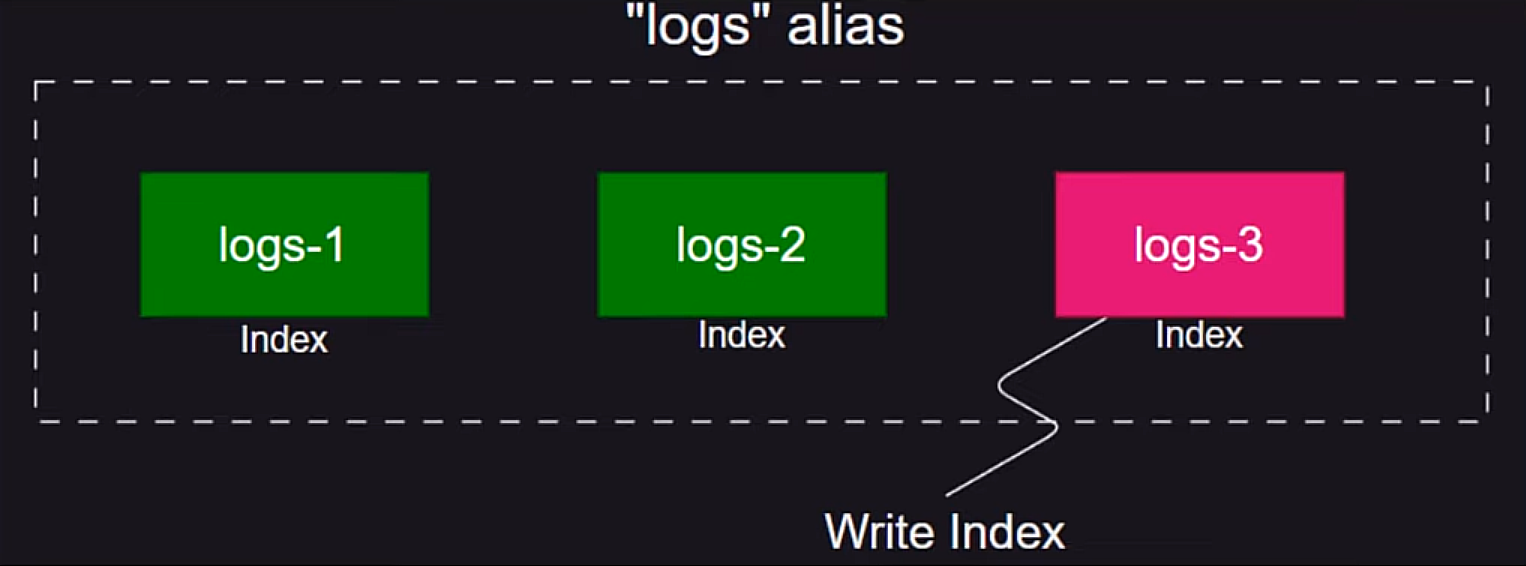
If in the next document there is a field 'Age' with a type string, Elasticsearch will think that the field 'Age' is a composite data type, where it can be either number or string.

If the index data mapping is set to strict, then for example, there is a schema that says that the field 'Age' is a number. If a document comes with a string value for that field, it will be rejected for not complying to the index data mapping properties(to the schema).

In DebugIt, Index template is set to strict, since classes' fields are mapped to indices in Program.cs. (fact check)

**Index alias** - grouping of Elasticsearch indices. If an index in the alias is a Write index, only then alias can have new documnts saved in it and the document will be saved in that Write index. Only one index in an alias can be the Write index.

It's useful to have index aliases when there's a huge amount of data incoming everyday and there's a need to split that data into smaller indices.



Example 5



Index alias can be used when there's a need to search through multiple indexes to find result for a query. In the example, all the indexes starting with 'logs-' will be in alias logs.

**Data streams** are for storing append-only time series data(data points recorded an organize in chronological order) across multiple indices while giving a single named resource for requests. Data streams are well-suited for logs, events, metric and other continuously generated data. ([more about data streams](https://www.elastic.co/guide/en/elasticsearch/reference/current/data-streams.html))

localhost:9200 - way to see if elasticsearch is running

Kibana - dashboard for Elasticsearch(read about it more later)

Fluent API for writing queries.

[About search](https://www.elastic.co/guide/en/elasticsearch/client/net-api/6.x/writing-queries.html)

By default, documents are returned in \_score descending order, where the \_score for each hit is the relevncy score calculated for how well the document matched th query criteria.

Users typically perform one of the following quries:

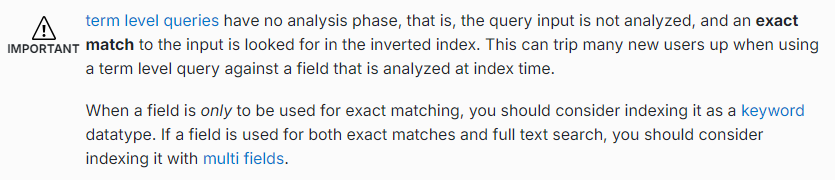
- structured search

- unstructured search

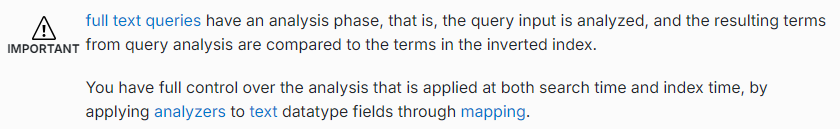
- combining queries

**Structured search** - it's about querying data that has inherent structures. Dates, times and numbers are all structured and it is common to want to query against fields of thee types to look for exact matches, values that fall within a raang, etc. Text can also be structured, for example, the keyword tags applied to a blog post.

With structured search, the answer to a query is always yes or no; a document is either a match for the query or it isn't, so there's no scoring the query. To do that, query can be executed in filter context by wrappipng it in a bool query filter clause. The benefit of executing a query in a filter context is that Elasticsearch is able to forgo calculating a relevancy score, as well as a cache filters for a faster subsequent performance.



**Unstructured search** - common use is to search within a full text fields in orer to find the most relevant documents. Full text queries are used for unstructured search.



**Combining queries** - extremely common scenario is to combine separate querie togther to form a compound query, the most common of which is the bool query

<https://www.red-gate.com/simple-talk/development/dotnet-development/how-to-build-a-search-page-with-elasticsearch-and-net/>

Internet users are used to Google-like search and they're expecting bspoke search t be a fast and precise. They need autocomplete, they assume that the search tolerats misspellings and they expect to be able to use filters and many other avnaced search features.

'Why would there be a need for other search engines when people are happy with SQL Server's Full-Text Search feature?'

- It might be enough for simple searches, but other search engines, like Elasticsearch, are a better choice when there's a ned to index and search unstructured data from different sources or when there's a need for custom functionality such as spellchecking, hit-highlightning, autocomplete or advanced scoring.

Elasticsearch is an open source search engine, written in Java and based on Lucene.

Stack Exchange initially grew on SQL Server Full-Text Search, but the limitations of it's feature and performance forced them to migrate to Elasticsearch for their search requirements.

**NEST** - one of two official .NET clients for interacting with Elasticsearch. It's a high-level client which maps closely to ElasticsearchAPI. All request and response objects are mapped. NEST provides the alternatives of either a fluent syntax for building queries, which resembels structure of raw JSON requests to API, or the use of object initializer syntax.

[Elasticsearch-github](https://github.com/rseniuta/elasticsearch-nest-webapi-angularjs)

**Bucket aggregations** - create buckets of documents. Each buckt is associated with a criterion(depending on the aggregation type) which determines whether or not a document in the current context 'falls' in it. In other words, the buckets effectively define document sets. In addition to the buckets themselves, the bucket aggregation also compute and return the number of ocument that 'fell into' each bucket.

They can hold sub-aggregations. These sub-aggregations will be aggregated for the buckets created by their 'parent' bucket aggregation.

There are different bucket aggregators, each with a different 'bucketing' strategy. Some define a single bucket, some define fixed numbr of multiple buckets and other dynamically create the buckets during the aggregation process.

[Metrics aggregations](https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-metrics.html)

[Analyzers](https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-analyzers.html#:~:text=The%20standard%20analyzer%20divides%20text,which%20is%20not%20a%20letter.)

Standard analyzer - default by Elasticsearch NEST

question-index - uses Serbian Latin tokenizer and analyzer so that search can be flexible for typing in Serbian Latin(provide image)

<https://opster.com/guides/elasticsearch/data-architecture/elasticsearch-text-analyzers/>

Elasticsearch prepares incoming text for efficient storing and searching. The text fields undergo Text Analysis process, where content of text fields is broken into individual toekns and are then enriched, transformed, synonymized, stemmed and captured in the inverted indices.

**Text analysis** - a process that iss taske with two functions:

- tokenization

- normalization

**Tokenization** - a process of splitting text content into indiviual words by inserting a whitespace delimiter, a letter, a pattern or a criteria. This process is carried out by a component called a tokenizer, whose sole job is to chop the content into individual words.

**Normalization** - a process where tokes(words) are transformed, modified and enriched in the form of stemming, synonym, stop words and other feautres. Stemming is an operation where the words are reduced(stemmed) to their root words: for example 'game' is a root word for 'gaming', 'gamer' including the plural form 'gamers'.

<https://www.elastic.co/guide/en/elasticsearch/reference/current/stemming.html>

Stemming is languag-dependent but often involvs removing prefixes and suffixes from words.

In some cases, the root form of a stemmed word may not be a real word (jumping and jumpiness have root jumpi), but that doesnt matter for search, if all variants of a word are reduced to the same root form, they will match correctly. Stemming is done in token filters.

Stemming token filters can be categorizd based on how they stem words:

- algorithmic stemmers

- dictionary stemmers

**Algorithmic stemmers** - apply a series of ruls to each word to reduce it to it's root form. For example, an algorithmic stemmer for English may remove -es and -s suffixes from the end of plural words. (DebugIt uses this stemmer)

Advantages:

- the require little setup and usually work well out of the box

- they use little memory

- they are typically faster than dictionary stemmers

However, most algorithmic stemmers only alter the exiting text of a word. This means they my not work well with irregular words that don't contain their root form, for example be, are and am or mouse and mice(in DebugIt koriscenje and koristiti).

**Dictionary stemmers** - look up words in a provied dictionary, replacing unstemmed word variants ith stemmed words from the dictionary. In theory, dictionary stemmers are well suited for:

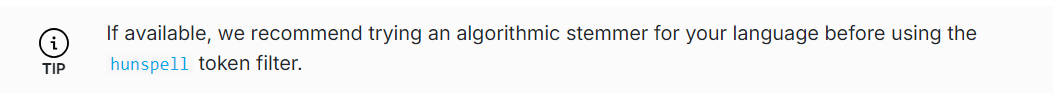
- stemming irregular words

- discerning betwen words tahta re spelled similarly but not related conceptually, such ass organ and organization

In practice, algoritmic stemmers typically outperform dictionary stemmers. This is because dictionary stemmers have the following disadventeges:

- Dictionary quality - a dictionary stemmer is only as good as its dictionary.To work well, these dictionaries must include a significent number of words, be updated regularly and change with language trends. Often, by the time a dictionary has been made available, it's incomplete and some of its entries are already outdated.

- Size and performance - dictionary stemmers must load all words, prefixes and suffixes from it's dictionary into memory. This can use a significant amount of RAM. Low-quality dictionaries may also be less efficient with prefix and suffix removal, which can slow the stemming process significantly.



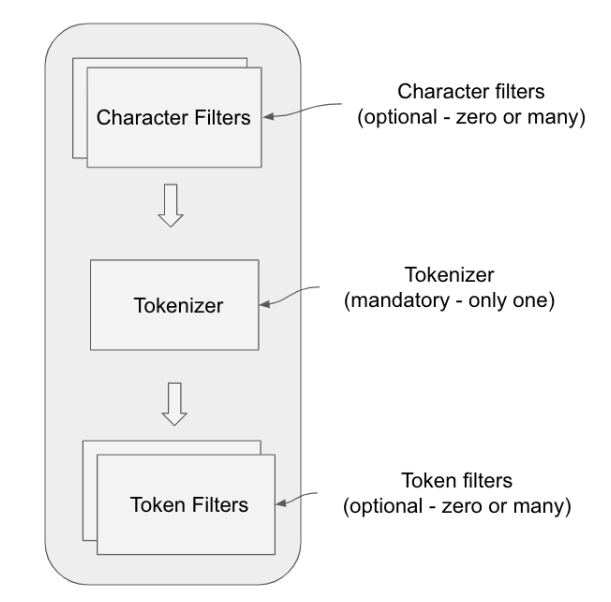
Sometimes stemming can prouce shared root words that ar spelled similarly, but not related conceptully, for example, a stemmer may reduces bot skies and skiing to the same root word ski. In that case, ther are several toke filters that can be used to prevent that and that can provide better control stemming.

Stemming changes tokens, so the same stemmer token filters should be used during index and search analysis.

Index analysis - when a document is indexed, any text field values are analyzed

Search analysis - when running a full-text search on a text field, the query string is analyzed

Text analysis is carried out by analyzers. Both of these funcntions are carrioud out by the analyzer module. An analyzer does this by using one tokenizer and zero or more token filters. In DebugIt, analyzer is language-specific analyzer.



An analyzer consists of three componenst:

- character filters(in DebugIt there are 0)

- tokenizer(in DebugIt there is 1)

- token filters(in DebugIt there are 2)

These components form a pipeline that each text field passes through. Character and token filters are optionaly, tokenizer is the only mandatory component and ther must be only one.

**Character filters** - remove unwantd characters from the input text string. Elasticsearch provide thre-character filtrs out of the box:

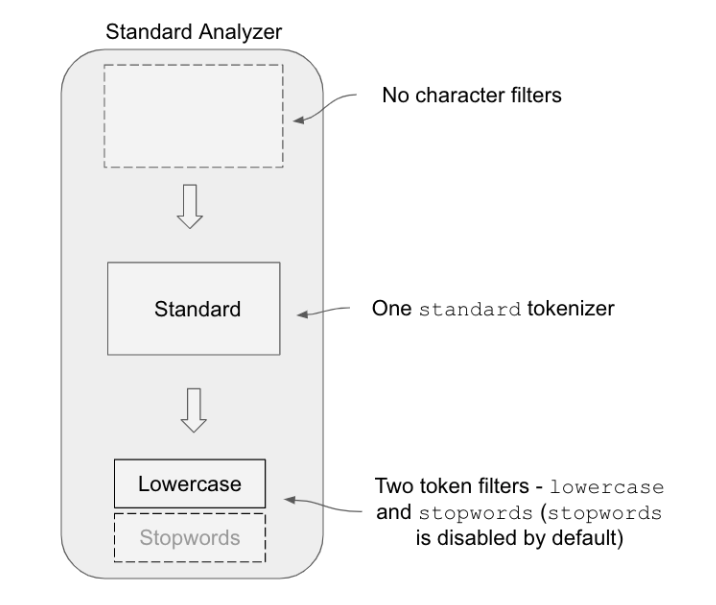
- html\_strip

- mapping

- pattern\_replace

**Tokenizers** - split the text into words by using delimiters such as whitespace, punctuation or some form of word boundaries. Elasticsearch uses standard tokenizer that splits the words based on grammar and punctuation, but thr are a lot of other tokenizers such as keyword, pattern, whitespace, lowercase etc.

**Token filters** - work on tokens from tokenizer for further processing. Tokens can change the case, create synonyms, provide the root word(steeming) etc.



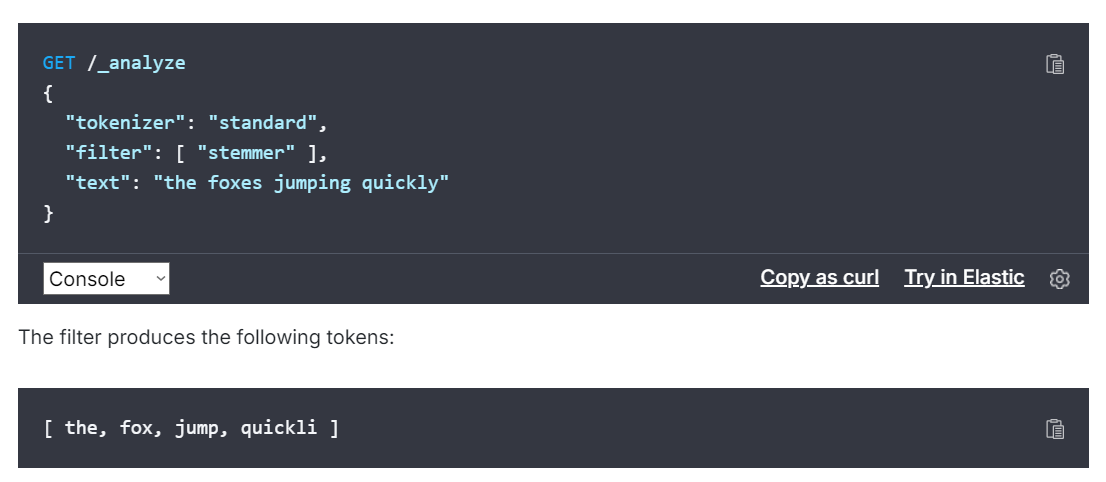
**Standard analyzer** - default analyzer whose job is to tokenize sentences based on whitespaces, punctuation and grammar. The pipeline consists of no character filters, one standard tokenizer and two filtrs: lowercase and stop words. Stop words token filtr is disabled by default.

Stop words - words commonly used in a language, but are usuallay filtered out in text analysis because they give little value in understanding the meaning of a sentence. They are often removed to focus on the more meaningful nd contextually relevant terms for more effective language processing.

[Stop words for Serbia taken from](https://github.com/stopwords-iso/stopwords-hr)

Example 6

In the pictures is an example of how stemming changes words.



Stemming - Elasticsearch's built in stemming for Serbian, but doesnt recognized all the root words(koriscenje, koristiti should have the same root, but they dont)

[Scoring](https://opster.com/guides/elasticsearch/search-apis/elasticsearch-scoring-understanding-the-explain-api/#:~:text=Scoring%20Mechanisms%20in%20Elasticsearch,%2C%20and%20field-length%20normalization.)

Elasticsearch is a powerful engine that provides fast and relevant search results by calculating a score for each document in the index. This score is a crucial factor in determining the order of the search results.

**Scoring mechanisms in Elasticsearch**

Elasticsearch uses a scoring model called the Practical Scoring Function (BM25) by default. This model is based on the probabilistic information retrieval theory and takes into account factors such as term frequency, inverse document frequency and fild-length normalization.

**Term frequency** - represents the number of times a term appears in a document. A higher term frequency indicates a stronger relationship between the term and the document.

**Inverse document frequency** - measures the importance of a term in the entire document collection. A term that appears in many documents is considered less important, while a term that appears in fewer documents is considered more important.

**Field-length normalization** - accounts for the length of the field in which the term appears. Shorter fields are given morer weight, as the term is considered more significant in a shorter field.(in DebugIt, there is Tags field used for search).

Explain API - valuable tool for understanding the scoring process. It provides a detailed explanation of how the core for a specific document was calucaletd. To use it, there has to be a GET request sent to the endpoint:

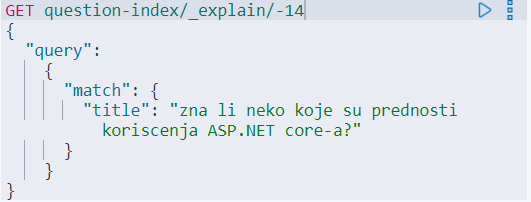


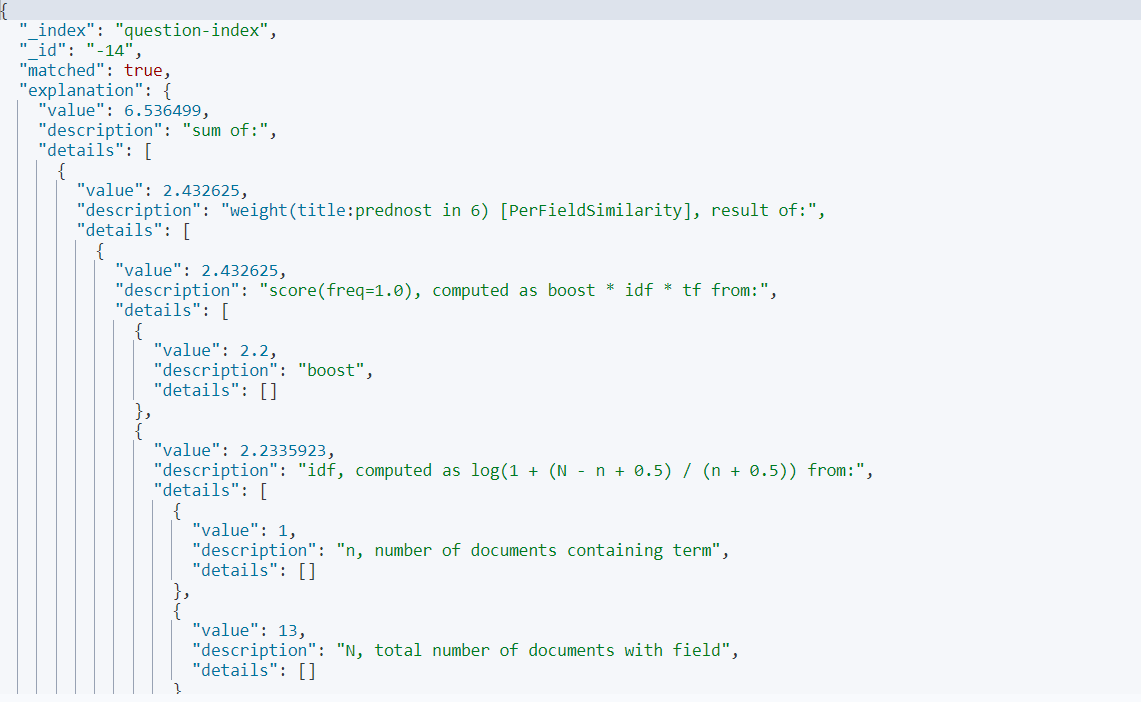
In the request body, a query must be provided for which there's a need for an explanation of the scoring. Example:



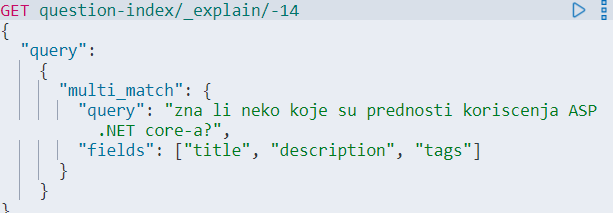
Explanation from Explain API will include a detailed breakdown of the scoring process, including the individual factors(TF, IDF and field-length normalization) and theri contributions to the final score.

The detailed explanation helps in understanding the factors contributing to the score and can be useful for fine-tuning the search relevance.





In DebugIt, search is done on multiple fields:





[Stack Overflow site for Elastic](https://discuss.elastic.co)

Scores are relative, so there is no max defined and it cannot be set either. Score is relevant to other documents, so there shouldn't be forcing the range of score to fit the specific range, because that would be changing the results of the search.

Example 7

There are 2 searches run against the same index. The range of relevance scores from the first search range from 0.0001 through 1.5000 and that is scaled so that it's between 0 and 1. For the second search, suppose the relevance score range is between 0.0500 and 3.5000 and it's scaled so that it's between 0 and 1. Now there are two sets of result and since both sets have the highest score of search 1, there could be the idea where these two results have the same value, but they dont. Since they're scaled, there's a loss of original relative scorings that would enable comparing the top result of each of the searches.

[Elasticsearch official site](https://www.elastic.co)

Sending 'C#' in a query doesnt work properly, since '#' is a reserved character in URLS, it needs to be encoded using encodeURIComponent().

**Search with synonyms**

[Search with synonyms](https://www.elastic.co/guide/en/elasticsearch/reference/current/search-with-synonyms.html)

Using synonyms in aspect of search is important, as they can imrpove search experience and increase the scope of search results.

Synonyms allow to:

- improve search relevance - by finding relevant documents taht use different terms to express the same concept

- make domain-specific vocabulariy - more user-friendly, allowing users to use search terms they are more familiar with

- define comoon misspellings and typos - to transparently handle common mistakes

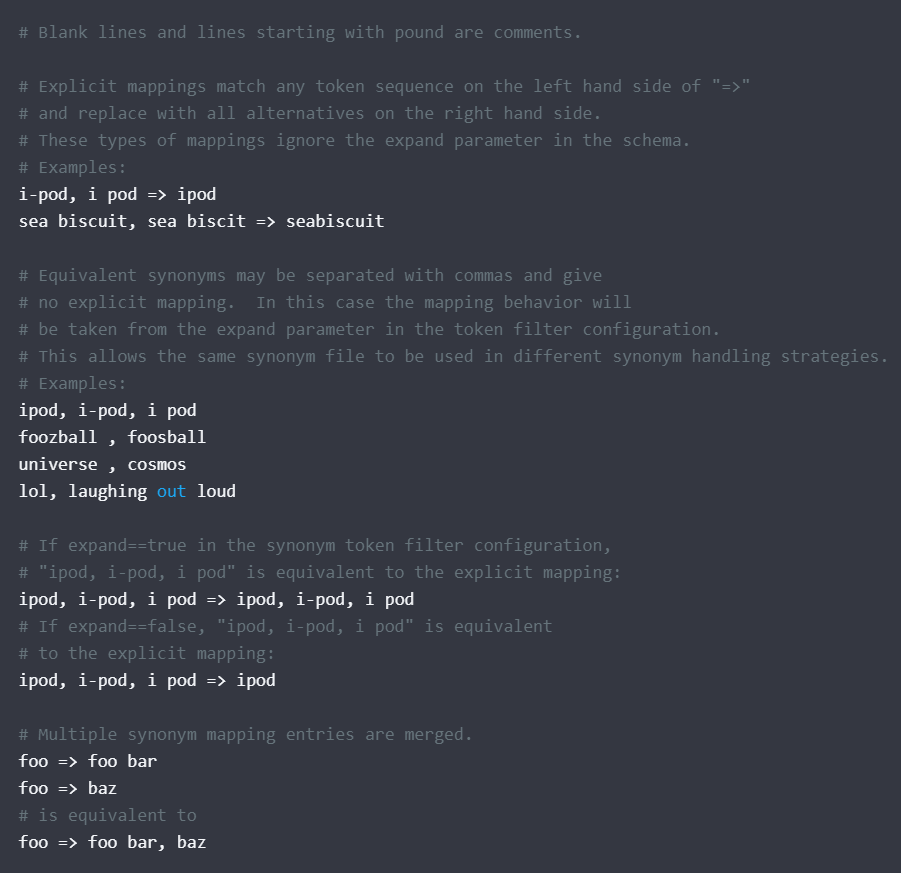
Synonyms are grouped together using synonym sets. There can be as many synonym sets as needed.

In order to use synonym sets in Elasticsearch, they need to be **stored** and **synonyms token filters and analyzers need to be configured**.

Synonyms sets need to be stored in Elasticsearch so analyzers can refer to them. There are 3 ways to store them:

- synonyms API - most flexible approach, as it allows to dynamically define and modify synonyms sets. Changes in synonyms sets will automatically reload the associated analyzers.

- synonyms file - needs to uploaded to all cluster nodes and be located in the configuration direcotroy for Elasticsearch distribution. (example on site). To update existing synonyms set, new files are uploaded to cluster. Synonyms set files must be kept in sync on every cluster node. When a synonyms set is updated, search analyzers that use it need to be refreshed. Manual syncing and reloading make this approach less flexible than using the synonyms API



- inline - adding synonyms directly inline in token filter definition. Not recommended for production usage. A large number of inline synonyms increases cluster size unnecessarily and can lead to performance issues.

After creating synonyms sets, token filters and analyzers can be configured to use them.

Synonyms sets must exist before they can be added to indices. If an index is created referencing a nonexistent synonyms set, the index will remain in a partially created and inoperable state. The only way to recover from this scenario is to ensure the synonyms set exists then either delete and re-create the index, or close and re-open the index.

Invalid synonym rules can cause errors when applying analyzer changes. For reloadable analyzers, this prevents reloading and applying changes. Errors in the synonym rules muust be corrected and anaylzer can be reloaded then.

An index with invalid synonym rules cannot be reopened, making it inoperable when:

- a node containing the index starts

- the index is opened from a closed state

- a node restart occurs (which reopens the node assigned shards)

Elasticsearch uses synonyms as part of the **analysis process**. There are 2 types of token filters that can be used to include synonyms:

- synonym graph - recommended, as it can correctly handle multi-word synonyms

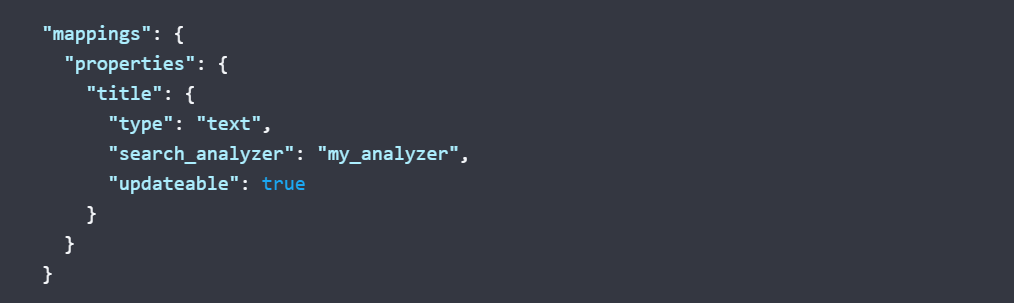
- synonym - not recommended if theres a need to use mulit-word synonyms

Synonyms can be applied at index or search time.

**Index time** - synonyms are applied when the documents are indexed into Elasticsearch. This is a less flexible alternative, as changes to your synonyms require reindexing.

**Search time** - synonymes are applied when a search is executed. This is a more flexible approach, which doesn't require reindexing. If token filters are configured with "updateable": true, search analyzers can be reloaded when you make changes to your synonyms. Synonym sets created using synonyms API can only be used at search time.

Analyzer that contains synonyms can be set as a search time or as an index time analyzer.



my\_analyzer added as a search analyzer to the title field in an indexing mapping

**Synonyms APIs**

Synonym sets can be used to configure **synonym graph token filters** and **synonym token filters**. These filters are applied as part of the analysis process by the search analyzer.

Synonym sets are limited to a maximum of 10 000 synonym rules per set.

Synonym rule - defines a group of words that act as synonyms. Individual synonym rules of a synonyms set can be updated without updating the whole synonyms set.

No support for synonyms APIs in NEST.

Fuzzy logic - a mathematics logic in which the truth of variables migth be any number between 0 and 1. It is different with a Boolean logic that only has the truth values either 0 or 1.

In the Elasticsearch, fuzzy query means the terms in the queries dont have to be the exact match with the terms in the Inverted index.

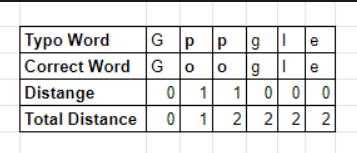
To calculate teh distance between query, Elasticsearch uses Levenshtein Distance Algorithm.

[Fuzzy search in Elasticsearch](https://codecurated.com/blog/how-to-handle-typos-in-elasticsearch-using-fuzzy-query/#:~:text=Handling%20a%20typo%20in%20Elasticsearch,fuzziness”%20being%20the%20most%20important.) (check for examples)

Calculating distance using Levenshtein Distance Algorithm(Levenshtein Edit Distance):

The first and second word are compared character by character. If the character is different, then the distance can be added between the words by one.

Example 8



Using Elasticsearch fuzzy query, Levenshtein Distance Algorithm will give output 2 for these 2 words.

If "fuzziness" is set to "AUTO" in a query, Elasticsearch will determine what fuzziness distance is appropriate. For 6 characters, the Elasticsearch will by default allow 2 edit distances. "AUTO" fuzziness is preferable, but it can also be tuned with an exact number when needed.



This is when fuzzy query is used as a parameter inside Match Query.

Fuzzy feature can be used as a Fuzzy Query.

Fuzzy Query - works just like Term Query, the query to Elasticsearch is not analyzed and used raw to serach the Inverted Index.

There are two values that can be passed for fuzziness parameter, an integer for exact maximum distance and "AUTO".

There are 2 parameters in the "AUTO" value and it looks like "AUTO:[low]:[high]". The query will set fuzziness as 0 if the term length is below the low value. If the term length is between the low and high value, the query will set the fuzziness to 1, but if the term length is more than the high value, the query will set the fuzziness to 2. (read the explanation from page 21)

The Elasticsearch will use 3 and 6 as default if the low and high values are not determined.

Match Query with fuzziness parameter is more preferable than Fuzzy Query.

**Transposition** - allows query to calculate the transpositions of tow adjacent characters (ab -> ba) as 1 distance. If "transpoistions" is set to true, word leasticsearch will have only distance of 1 compared to elasticsearch. It cannot be set in Match Queries as transpositions will be caculated as 1 distance

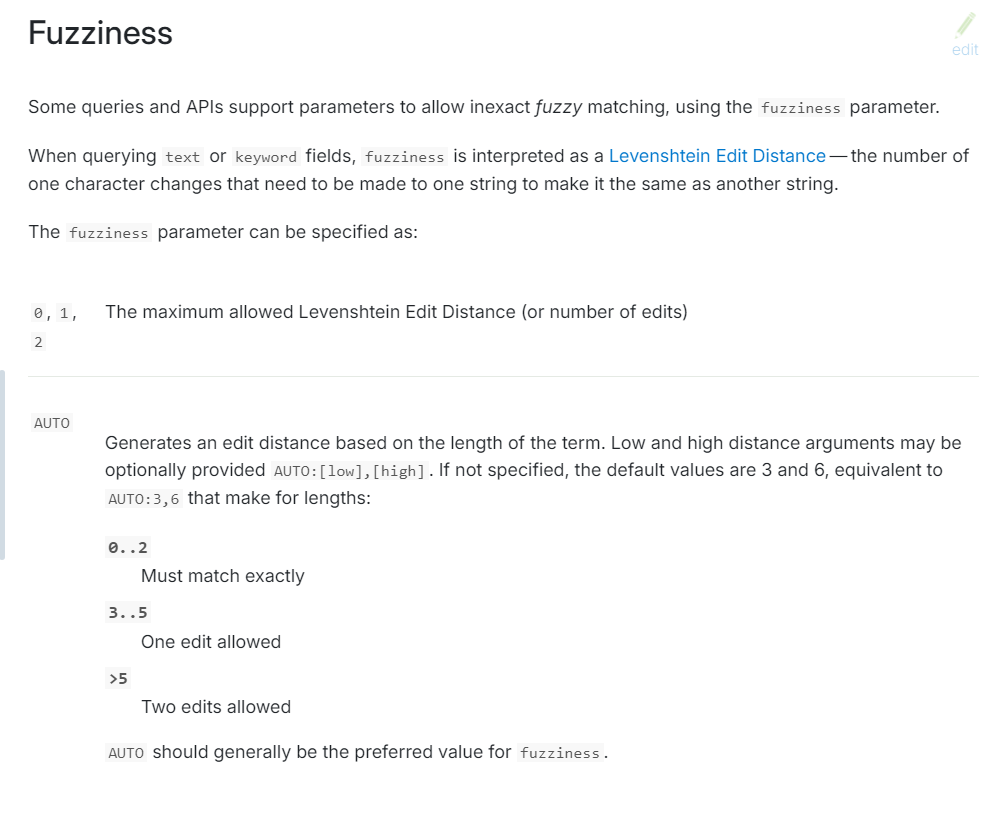


**Max expansions** - determines the maximum result got from query. If its set to 1 and there are 2 documents that are appropriate to the query, Elasticsearch will only return 1. It's applied to shard level, so if there are many shards, even if "max\_expansions" is set to 1, query might return more results. Default value for "max\_expansions" is 50.

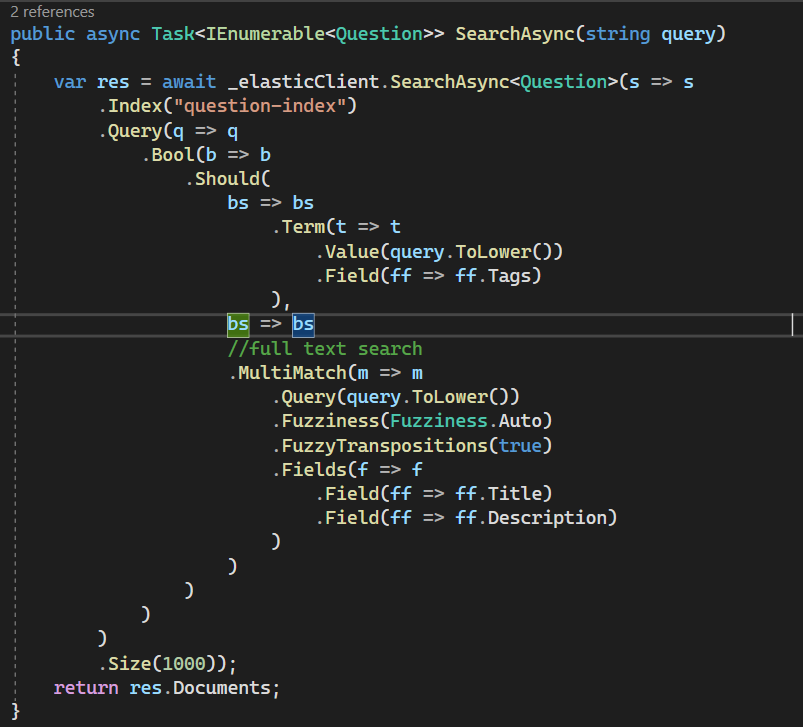
**Prefix length** - number of prefix characters that is not considered in fuzzy query. If "prefix\_length" is set to 1, there will be no results if word 'llasticsearch" is queried. It's default value is 0.

**Rewrite** - used for changing the scoring of the results.

[Official Elasticsearch site for fuzzy query](https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-fuzzy-query.html)



In DebugIt, search will not be like this:



Because users need to be able to have typos and find questions based on tags as well. With .Term, search will return documents that contain an exact term in a provided field. [Term query explanation](https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-term-query.html)

[More about fuzzy search](https://dev.to/hernamvel/understanding-and-tuning-fuzzy-queries-in-elasticsearch-by-example-1ci3#:~:text=Undestanding%20"fuziness"%3A%20"auto"&text=In%20this%20sense%2C%20ElasticSearch%20allows,exact%20matches%20will%20be%20returned.)

Levenshtein Edit Distance Algorithm is defined as the number of individual changes required to make one term identical to the another one. A change can be an insert, edit or delete operation.

Executing a fuzzy query is not that far from executing a typical search query in Elasticsearch. It's important to add the fuzziness parameter in the JSON request with the maximum Levenshtein distance that is allowed.

Sometimes there's a need for more flexibility on how long the maximum of Levenshtein Distance should be. For short terms, it makes no sense to have a distance greater than 2 because it can generate a lot of false positives. In this sense, Elasticsearch allows the value "auto" to the fuzziness parameter that will work in this way.

If the length of the term is 0, 1 or 2, the maximum Levenshtein distance allowed will be 0. In other words, only terms with exact matches will be returned.

If the length of the term is 3, 4 or 5, the maximum Levenshtein distance allowed will be 1.

If the length of the term is greater than 5, the maximum Levenshtein distance allowed will be 2.

The above explanation is not supported in NEST.