Contents

[Types of Query 2](#_Toc51111432)

[query context and the filter context 2](#_Toc51111433)

[Query context 3](#_Toc51111434)

[Filter context 3](#_Toc51111435)

[Text vs. keyword 3](#_Toc51111436)

[Query DSL 3](#_Toc51111437)

[Structured querying 4](#_Toc51111438)

[Term query 5](#_Toc51111439)

[Term query with a filter 5](#_Toc51111440)

[Terms query 6](#_Toc51111441)

[Terms set query 6](#_Toc51111442)

[Range query 6](#_Toc51111443)

[Exists query 7](#_Toc51111444)

[Full-text querying 7](#_Toc51111445)

[Match All Query 8](#_Toc51111446)

[Match query 9](#_Toc51111447)

[Match phrase query 9](#_Toc51111448)

[Multi-match query 10](#_Toc51111449)

[Compound queries 10](#_Toc51111450)

[Boolean query 11](#_Toc51111451)

[Boosting query 11](#_Toc51111452)

[Constant score query 11](#_Toc51111453)

[Aggregations 11](#_Toc51111454)

[bucket aggregations 11](#_Toc51111455)

[pipeline aggregations 11](#_Toc51111456)

[metric aggregations 12](#_Toc51111457)

[matrix aggregations 12](#_Toc51111458)

# Types of Query

<https://medium.com/elasticsearch/introduction-to-elasticsearch-queries-b5ea254bf455>

The queries in Elasticsearch can be broadly classified into two categories,

1. The leaf queries: Leaf queries look for specific values in certain field/fields. These queries can be used independently. Some of these queries include match, term, range queries. <https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html>
2. The compound queries: Compound queries uses the combination of leaf/compound queries. Basically, they combine multiple queries together to achieve their target results. <https://www.elastic.co/guide/en/elasticsearch/reference/current/compound-queries.html>

## query context and the filter context

Elasticsearch, by default, while returning the search results, would sort them based on their relevance score, which indicates how well the document matches the query. This relevance score is computed and returned in the **\_score** parameter of the metadata with each result.

This is by default a positive floating point number.

The **\_score** computation techniques might be different for different type of queries.

The score computation depends on what context the query clause was run i.e. the **query** clauses were run in the “**query**” context or in the “**filter**” context.

### Query context

When a clause is executed in the **query** context, it looks for “**how well the document matched to the query**”. The more the match, the higher the score would be.

### Filter context

When the query clause is given in the **filter** context, it just looks if the documents contains the clause of not. That is effectively a true/false returning. Suppose we are querying the data in the filter context, by asking whether the document field gender matches “Male”, we will get only the matching documents, with no score.

**Unlike the query context, the filter context does not use time to compute the scores, and hence filter context return faster results.**

## Text vs. keyword

<https://www.elastic.co/blog/strings-are-dead-long-live-strings>

Elasticsearch has two very different ways to search strings. You can either search whole values, that we often refer to as keyword search, or individual tokens, that we usually refer to as full-text search. If you are familiar with Elasticsearch, you know the former strings should be mapped as a not\_analyzed string while the latter should be mapped as an analyzed string.

strings will now be mapped both as text and keyword by default. text, which should be used for full-text search, and keyword, which should be used for keyword search and aggregation.

# Query DSL

<https://towardsdatascience.com/deep-dive-into-querying-elasticsearch-filter-vs-query-full-text-search-b861b06bd4c0>

<https://medium.com/@User3141592/understanding-the-elasticsearch-query-dsl-ce1d67f1aa5b>

An over-arching theme in **Elasticsearch** is that all **queries** can be classified into three types:

1. **Filtering by exact values**2. **Searching on analyzed text**  
3. **A combination of the two**

All query clauses have either one of these two formats:

{  
 QUERY\_CLAUSE: {  
 ARGUMENT: VALUE,  
 ARGUMENT: VALUE,...  
 }  
}{  
 QUERY\_CLAUSE: {  
 FIELD\_NAME: {  
 ARGUMENT: VALUE,  
 ARGUMENT: VALUE,...  
 }  
 }  
}

The syntax-rule is that query clauses can be repeatedly nested inside other query clauses

{  
 QUERY\_CLAUSE {  
 QUERY\_CLAUSE: {  
 QUERY\_CLAUSE: {  
 QUERY\_CLAUSE: {  
 ARGUMENT: VALUE,  
 ARGUMENT: VALUE,...  
 }  
 }  
 }  
 }  
}

The rule of a thumb would be to use **filters** for:

* yes/no search
* search on exact values (numeric, range and keyword)

Use **queries** for:

* ambiguous result (some documents suit more than others)
* full-text search

## Structured querying

Also called **term-level queries**, structured queries are a group of querying methods that checks if a document should be selected or not. Therefore, there is no real need for relevance score in many cases — document either going to match or not (especially numerics).

**Term-level queries are still queries, so they will return the score**.

### Term query

Returns the documents where the value of a field exactly matches the criteria. The term query is somewhat an alternative of SQL

**select \* from table\_name where column\_name =...**

The term query goes directly to the inverted index which makes it fast. It is preferred to use term only for keyword fields when working with text data.

GET /\_search  
{  
 "query": {  
 "term": {  
 "<field\_name>": {  
 "value": "<your\_value>"  
 }  
 }  
 }  
}

**The term query is run in the query context by default, therefore, it will calculate the score.** Even if the score will be identical for all documents returned, additional computing power will be involved.

### Term query with a filter

If we want to speed up term query and get it cached then it should be wrapped up in a **constant\_score** filter.

Remember the rule of thumb? Use this method if you do not care about the relevance score.

GET /\_search  
{  
 "query": {  
 "**constant\_score**" : {  
 "**filter**" : {  
 "term" : {"<field\_name>" : "<your\_value>"}  
 }  
 }  
 }  
}

Now, the query is not calculating any relevance score, therefore, it is faster. Moreover, it is automatically cached.

**Quick advise — use match instead of term for text fields.**

Remember, the term query goes directly to the inverted index. Term query takes the value you provide and searches for it as it is, that is why it suits well for querying **keyword** fields that are stored without any transformations.

### Terms query

As you could have guessed, the term query allows you to return documents which are matching at least one exact term.Term query is somewhat an alternative of

**SQL select \* from table\_name where column\_name is in...**

Important to understand that querying field in Elasticsearch might be a list, for example

**{ "name" : ["Odin", "Woden", "Wodan"] }.**

If you perform a terms query that contains one of the following names then this record will be matched — it does not have to match all the values in the field, but only one.

GET /\_search  
{  
 "query" : {  
 "terms" : {  
 "name" : ["Frigg", "Odin", "Baldr"]  
 }  
 }  
}

### Terms set query

Same as terms query but this time **you can specify how many exact terms should be in the queried field.**

You specify how many have to match — one, two, three or all of them. However, this number is another numeric field. Therefore, each document should contain this number (specific to this particular document).

### Range query

Returns documents in which queried field’s value is within the defined range.Equivalent of SQL

**select \* from table\_name where column\_name is between...**

Range query has its own syntax:

* **gt** is greater than
* **gte** is greater than or equal to
* **lt** is less than
* **lte** is less than or equal to

An example where the field’s value should be ≥ 4 and ≤ 17:

GET \_search  
{  
 "query": {  
 "range" : {  
 "<field\_name>" : {  
 "gte" : 4,  
 "lte" : 17  
 }  
 }  
 }  
}

The range query also works well with dates.

### Exists query

Due to the fact that Elasticsearch is schemaless (or no strict schema limitation), it is a fairly common situation when different documents have different fields. As a result, there is a lot of use to know whether a document has any certain field or not.

**Exists query returns documents that contain an indexed value for a field**

GET /\_search  
{  
 "query": {  
 "exists": {  
 "field": "<your\_field\_name>"  
 }  
 }  
}

## Full-text querying

Full-text queries work well with unstructured text data**. Full-text queries take advantage of the analyzer.**

**Elasticsearch’s analyzer pipe**

Every time text type data is inserted into the Elasticsearch index it is analyzed and, then, stored at the inverted index. Depending on how you configure the analyzer will impact your searching capabilities because analyzer is also applied for full-text search.

***Analyzer pipe*** consists of three stages:

Character filter (0+) → Tokenizer (1) → Token filter (0+)

There is always **one tokenizer** and **zero or more character & token filters**.

1. ***Character filter*** receives the text data as it is, then it might preprocess the data before it gets tokenized. Character filters are used to:

* Replace characters matching given regular expression
* Replace characters matching given strings
* Clean HTML text

1. ***Tokenizer*** breaks text data received after character filter (if any) into tokens. For example, whitespace tokenizer simply breaks text by the whitespace (it is not the standard one). Therefore, Wednesday is called after Woden. will be split into [Wednesday, is, called, after, Woden.]. There are many [build-in tokenizers](https://www.elastic.co/guide/en/elasticsearch/reference/7.x/analysis-tokenizers.html) that can be used to create custom analyzers.

***Standard tokenizer*** breaks text by whitespace after removing the punctuation. It is the most neutral option for the vast majority of languages.

In addition to tokenization, ***tokenizer*** does the following:

* keeps track of tokens order,
* notes start and end of each word
* defines the type of token

1. ***The token filter*** applies some transformation on the tokens. There are many different token filters that you might choose to add to your analyzer. Some of the most popular are:

* lowercase
* stemmer (exist for many languages!)
* remove duplicate
* transformation to the ASCII equivalent
* workaround with patterns
* limit on token count
* stop list of tokens (removes tokens from the stop list)

The **standard analyzer** is the default one**. It has 0 character filters, standard tokenizer, lowercase and stops token filters.** You can compose your custom analyzer as you wish, but there are also few build-in analyzers.

Some of the most efficient out of a box analyzers are the language analyzers that are taking the specifics of each language to make a more advanced transformation.

**The full-text query will use the same analyzer that was used while indexing the data.** More precisely, the text of your query will go through the same transformations as the text data in the searching field, so that both are at the same level.

### Match All Query

The **match all** query clause returns all documents. It’s analogous to

**SELECT \* FROM table**

GET /\_search  
{

"match\_all": {}

}

### Match query

**Match query is the standard query for querying the text fields.**

We might call match query an equivalent of the term query but for the text type fields (while term should be used solely for the keyword type field when working with text data).

GET /\_search  
{  
 "query" : {  
 "match" : {  
 "<text\_field>" {  
 "query" : "<your\_value>"  
 }  
 }  
 }  
}

The string that is passed into the **query** parameter (required one), by default, going to be processed by the same analyzer as the one that has been applied to the searched field. Unless you specify the analyzer yourself using **analyzer** parameter.

When you specify your phrase to be searched for it is being analyzed and the result is always a set of tokens. By default, Elasticsearch will be using **OR** operator between all of those tokens. That means that at least one should match — more matches will hit a higher score though. You might switch this to **AND** in **operator parameter**. In this case, **all of the tokens will have to be found in the document for it to be returned.**

If you want to have something in between **OR** and **AND** you might specify **minimum\_should\_match** parameter which specifies the number of clauses that should match. It can be specified in both, number and percentage.

**fuzziness** parameter (optional) allows you to omit the typos. **Levenshtein** distance is used for calculations.

If you apply **match** query to the **keyword** field then it will perform the same as **term** query. More interestingly, if you pass the exact value of a token that is stored in an inverted index to the **term** query then it will return exactly the same result as **match** query but faster as it will go straight to the inverted index.

### Match phrase query

Same as **match** but the sequence order and proximity are important. **Match** query is not aware of the sequence and proximity, therefore, it is only possible to achieve the phrase match with a different type of a query.

GET /\_search  
{  
 "query": {  
 "match\_phrase" : {  
 "<text\_field>" : {  
 "query" : "<your\_value>",  
 "slop" : "0"  
 }  
 }  
 }  
}

**match\_phrase** query has **slop** parameter (default value 0) which is responsible for skipping the terms. Therefore, if you specify **slop** equal to 1 then one word out of a phrase might be omitted.

### Multi-match query

**Multi-match** query does the same job as the **match** with the only difference that it is applied to more than one field.

GET /\_search  
{  
 "query": {  
 "multi\_match" : {  
 "query": "<your\_value>",   
 "fields": [ "<text\_field1>", "<text\_field2>" ]   
 }  
 }  
}

* **fields** names can be specified using wildcards
* each field is equally weighted by default
* each field’s contribution to the score can be boosted
* if no fields specified in the fields parameter then all eligible fields will be searched

There are different types of multi\_match.

There are [different types](https://www.elastic.co/guide/en/elasticsearch/reference/7.x/query-dsl-multi-match-query.html) of multi\_match. For example :

* **best\_fields** type (default) prefers results where tokens from searched value are found in one field to those results where searched tokens are split among different fields.
* **most\_fields** is somewhat opposite to best\_fields type.
* **phrase** type behaves as **best\_fields** but searches for the entire phrase similar to **match\_phrase**.

## Compound queries

Compound queries wrap together other queries. Compound queries:

* combines the score
* changes behavior of wrapped queries
* **switch query context to filter context**
* any of above combined

### Boolean query

[**Boolean query**](https://www.elastic.co/guide/en/elasticsearch/reference/7.x/query-dsl-bool-query.html)**combines together other queries.** It is the most important compound query.

**Boolean query allows you to combine searches in query context with filter context searches**.

The boolean query has four occurrences (types) that can be combined together:

* **must** or “**has to satisfy the clause**”
* **should** or “**additional points to relevance score if clause is satisfied**”
* **filter** or “**has to satisfy the clause but relevance score is not calculated**”
* **must\_not** or “**inverse to must, does not contribute to relevance score**”

**must** and **should** → **query context**

**filter** and **must\_not** → **filter context**

For those who are familiar with SQL **must** is **AND** while **should** is **OR** operators. Therefore, each query inside the must clause has to be satisfied.

### Boosting query

Boosting query is alike with boost parameter for most queries but is not the same. Boosting query returns documents that match positive clause and reduces the score for the documents that match negative clause.

### Constant score query

As we previously saw in term query example, **constant\_score** query **converts any query into filter context** with relevance score equal to the boost parameter (default 1).

Notes:

1. Do you really need to score your documents while querying?
2. Unless you need relevance score or full-text search always try to use filters. Filters are “cheaper”.
3. Filters are automatically cached and do not contribute to the relevance score.

# Aggregations

### bucket aggregations

### pipeline aggregations

### metric aggregations

### matrix aggregations