Query string syntax

edit

The query string “mini-language” is used by the Query String Query and by the `q` query string parameter in the `search` API.

The query string is parsed into a series of *terms* and *operators*. A term can be a single word — `quick` or `brown` — or a phrase, surrounded by double quotes — `"quick brown"` — which searches for all the words in the phrase, in the same order.

Operators allow you to customize the search — the available options are explained below.

Field names

edit

As mentioned in Query String Query, the `default_field` is searched for the search terms, but it is possible to specify other fields in the query syntax:

- where the `status` field contains `active`

```
status:active
```

- where the `title` field contains `quick` or `brown`. If you omit the OR operator the default operator will be used

```
title:(quick OR brown)
title:(quick brown)
```

- where the `author` field contains the exact phrase `"john smith"`

```
author:"John Smith"
```

- where any of the fields `book.title`, `book.content` or `book.date` contains `quick` or `brown` (note how we need to escape the `*` with a backslash):

```
book.\*:(quick brown)
```

- where the field `title` has no value (or is missing):

```
_missing_:title
```

- where the field `title` has any non-null value:

```
_exists_:title
```

Wildcards

edit

Wildcard searches can be run on individual terms, using `?` to replace a single character, and `*` to replace zero or more characters:

```
qu?ck bro*
```

Be aware that wildcard queries can use an enormous amount of memory and perform very badly—just think how many terms need to be queried to match the query string `"a* b* c*"`.

WARNING

Allowing a wildcard at the beginning of a word (eg `"*ing"`) is particularly heavy, because all terms in the index need to be examined, just in case they match. Leading wildcards can be disabled by setting `allow_leading_wildcard` to `false`.

Wildcarded terms are not analyzed by default—they are lowercased (`lowercase_expanded_terms` defaults to `true`) but no further analysis is done, mainly because it is impossible to accurately analyze a word that is missing some of its letters. However, by setting `analyze_wildcard` to `true`, an attempt will be made to analyze wildcarded words before searching the term list for matching terms.

Regular expressions

edit

Regular expression patterns can be embedded in the query string by wrapping them in forward-slashes (`"/"`):

```
name:/joh?n(ath[oa]n)/
```

The supported regular expression syntax is explained in Regular expression syntax.

WARNING

The `allow_leading_wildcard` parameter does not have any control over regular expressions. A query string such as the following would force Elasticsearch to visit every term in the index:

```
/.*n/
```

Use with caution!

Fuzziness

edit

We can search for terms that are similar to, but not exactly like our search terms, using the “fuzzy” operator:

```
quikc~ brwn~ foks~
```

This uses the Damerau-Levenshtein distance to find all terms with a maximum of two changes, where a change is the insertion, deletion or substitution of a single character, or transposition of two adjacent characters.

The default *edit distance* is `2`, but an edit distance of `1` should be sufficient to catch 80% of all human misspellings. It can be specified as:

```
quikc~1
```

Proximity searches

edit

While a phrase query (eg `"john smith"`) expects all of the terms in exactly the same order, a proximity query allows the specified words to be further apart or in a different order. In the same way that fuzzy queries can specify a maximum edit distance for characters in a word, a proximity search allows us to specify a maximum edit distance of words in a phrase:

```
"fox quick"~5
```

The closer the text in a field is to the original order specified in the query string, the more relevant that document is considered to be. When compared to the above example query, the phrase `"quick fox"` would be considered more relevant than `"quick brown fox"`.

Ranges

edit

Ranges can be specified for date, numeric or string fields. Inclusive ranges are specified with square brackets `[min TO max]` and exclusive ranges with curly brackets `{min TO max}`.

- All days in 2012:

  ```
  date:[2012-01-01 TO 2012-12-31]
  ```

- Numbers 1..5

  ```
  count:[1 TO 5]
  ```

- Tags between `alpha` and `omega`, excluding `alpha` and `omega`:

  ```
  tag:{alpha TO omega}
  ```

- Numbers from 10 upwards

  ```
  count:[10 TO *]
  ```

- Dates before 2012

  ```
  date:{* TO 2012-01-01}
  ```

Curly and square brackets can be combined:

- Numbers from 1 up to but not including 5

  ```
  count:[1 TO 5}
  ```

Ranges with one side unbounded can use the following syntax:

```
age:>10
age:>=10
age:<10
age:<=10
```

NOTE

To combine an upper and lower bound with the simplified syntax, you would need to join two clauses with an `AND` operator:

```
age:(>=10 AND <20)
age:(+>=10 +<20)
```

The parsing of ranges in query strings can be complex and error prone. It is much more reliable to use an explicit `range` query.

Boosting

edit

Use the *boost* operator `^` to make one term more relevant than another. For instance, if we want to find all documents about foxes, but we are especially interested in quick foxes:

```
quick^2 fox
```

The default `boost` value is 1, but can be any positive floating point number. Boosts between 0 and 1 reduce relevance.

Boosts can also be applied to phrases or to groups:

```
"john smith"^2   (foo bar)^4
```

Boolean operators

edit

By default, all terms are optional, as long as one term matches. A search for `foo bar baz` will find any document that contains one or more of `foo` or `bar` or `baz`. We have already discussed the `default_operator` above which allows you to force all terms to be required, but there are also *boolean operators* which can be used in the query string itself to provide more control.

The preferred operators are `+` (this term **must** be present) and `-` (this term **must not** be present). All other terms are optional. For example, this query:

```
quick brown +fox -news
```

NOTE
NLP-PIER's default logical operator is AND, instead of Elasticsearch's default OR.
This same query in NLP-PIER would result in 'quick', 'brown', and 'fox' being required and an absence of the term 'news'

states that:

- `fox` must be present
- `news` must not be present
- `quick` and `brown` are optional — their presence increases the relevance

The familiar operators `AND`, `OR` and `NOT` (also written `&&`, `||` and `!`) are also supported. However, the effects of these operators can be more complicated than is obvious at first glance. `NOT` takes precedence over `AND`, which takes precedence over `OR`. While the `+` and `-` only affect the term to the right of the operator, `AND` and `OR` can affect the terms to the left and right.

> Rewriting the above query using `AND`, `OR` and `NOT` demonstrates the complexity:
>
> **`quick OR brown AND fox AND NOT news`**
> This is incorrect, because `brown` is now a required term.
>
> **`(quick OR brown) AND fox AND NOT news`**
> This is incorrect because at least one of `quick` or `brown` is now required and the search for those terms would be scored differently from the original query.
>
> **`((quick AND fox) OR (brown AND fox) OR fox) AND NOT news`**
> This form now replicates the logic from the original query correctly, but the relevance scoring bears little resemblance to the original.
>
> In contrast, the same query rewritten using the `match` query would look like this:
>
> ```
> {
>     "bool": {
>         "must":     { "match": "fox"         },
>         "should":   { "match": "quick brown" },
>         "must_not": { "match": "news"        }
>     }
> }
> ```

Grouping

edit

Multiple terms or clauses can be grouped together with parentheses, to form sub-queries:

```
(quick OR brown) AND fox
```

Groups can be used to target a particular field, or to boost the result of a sub-query:

```
status:(active OR pending) title:(full text search)^2
```

Reserved characters

edit

If you need to use any of the characters which function as operators in your query itself (and not as operators), then you should escape them with a leading backslash. For instance, to search for `(1+1)=2`, you would need to write your query as `\(1\+1\)\=2`.

The reserved characters are: `+ - = && || > < ! ( ) { } [ ] ^ " ~ * ? : \ /`

Failing to escape these special characters correctly could lead to a syntax error which prevents your query from running.

Watch this space

A space may also be a reserved character. For instance, if you have a synonym list which converts `"wi fi"` to `"wifi"`, a `query_string` search for `"wi fi"` would fail. The query string parser would interpret your query as a search for `"wi OR fi"`, while the token stored in your index is actually `"wifi"`. Escaping the space will protect it from being touched by the query string parser: `"wi\ fi"`.