

Elasticsearch IN ACTION

Radu Gheorghe
Matthew Lee Hinman
Roy Russo





Elasticsearch in Action

by Radu Gheorghe
Matthew Lee Hinman
Roy Russo

Appendix C

Copyright 2016 Manning Publications

brief contents

PART 11

- 1 ■ Introducing Elasticsearch 3
- 2 ■ Diving into the functionality 20
- 3 ■ Indexing, updating, and deleting data 53
- 4 ■ Searching your data 83
- 5 ■ Analyzing your data 118
- 6 ■ Searching with relevancy 148
- 7 ■ Exploring your data with aggregations 179
- 8 ■ Relations among documents 215

PART 2259

- 9 ■ Scaling out 261
- 10 ■ Improving performance 293
- 11 ■ Administering your cluster 340

appendix C

Highlighting

Highlighting indicates why a document results from a query by emphasizing matching terms, giving the user an idea of what the document is about, and also showing its relationship to the query, as shown in figure C.1.

Although figure C.1 is taken from DuckDuckGo, Elasticsearch offers highlighting functionality, too. For example, you can search for “elasticsearch” in get-together event titles and make that word stand out like this:

```
"title" : [ "Introduction to <em>Elasticsearch</em>" ],
```

To get such highlighting, you’ll need three things, and we’ll discuss them in detail in this appendix:

- A highlight part of your search request, which will go on the same level as query and aggregations
- A list of fields you want to be highlighted, like the event name or its description
- Highlighted fields included in `_source` or stored individually

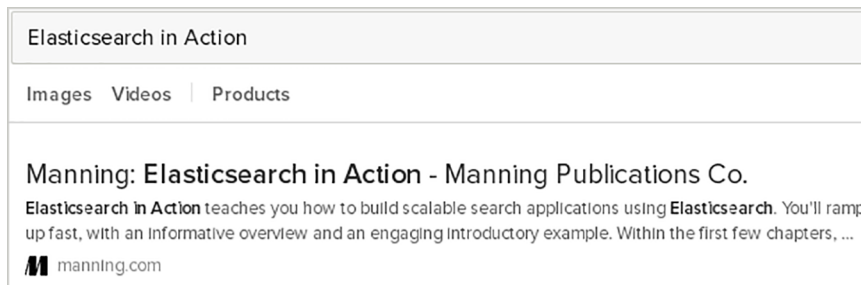


Figure C.1 Highlighting shows why a document matched a query.

NOTE All fields are included in `_source` by default but aren't stored individually. You can find more information about `_source` and stored fields in chapter 3, section 3.4.1.

After you do the basic highlighting, you might want to turn some knobs. In this appendix, we'll also discuss the most important highlighting options:

- *What to match*—You can decide, for example, to show a snippet of a field, even if there are no terms to highlight in there, to show the same fields for all documents. Or you might want to use a different query for highlighting than the one you use for searching.
- *How fragments should look*—With large fields, you typically don't get back all their contents with highlighted terms; you just get one or more fragments of text around those terms. You can configure how many fragments to allow, which order they should be shown, and how big they should be in.
- *How to highlight*—You can change the default `` and `` tags to something else. If you stick to HTML tags, you can have Elasticsearch encode the whole fragments in HTML (for example, by escaping ampersand (&) characters) so you can render those fragments correctly in your application.

We'll also discuss different highlighting implementations. The default implementation is called `plain` and relies on re-analyzing the text from stored fields in order to highlight relevant terms. This process might become too expensive for big fields, like the contents of a blog post. Alternatively, you can use the `Postings Highlighter` or the `Fast Vector Highlighter`. Both require you to change the mapping to make Elasticsearch store additional data: term offsets for the `Postings Highlighter` and term vectors for the `Fast Vector Highlighter`. Both changes will increase your index size and use more computing power while indexing.

Each highlighting implementation comes with its own set of features, and we'll talk about them later in this appendix. But first, let's deal with the basics of highlighting.

C.1 Highlighting basics

To start, you'll recreate the highlighting snippet from the introduction. In listing C.1, you'll run a search on the `get-together` events for the term “`elasticsearch`” in the `name` and will highlight this term in the `title` and the `description` fields.

NOTE For the listing to work, you need to download the code samples for this book by cloning the Git repository from <https://github.com/dakrone/elasticsearch-in-action> and running `populate.sh` to index the sample data.

Listing C.1 Highlighting terms in two fields

```
curl localhost:9200/get-together/event/_search?pretty -d '{
  "query": {
    "match": {
      "title": "elasticsearch"
    }
  }
}
```

Typical match query
for Elasticsearch;
nothing new here

```

    },
    "highlight": {
      "fields": {
        "title": {},
        "description": {}
      }
    }
  }
}
# reply
  "hits" : [ {
    "_index" : "get-together",
    "_type" : "event",
    "_id" : "103",
    "_score" : 0.9581454,
    "_source":{
      "host": "Lee",
      "title": "Introduction to Elasticsearch",
      "description": "An introduction to ES and each other. We can meet and
➤ greet and I will present on some Elasticsearch basics and how we use it.",
      [...]
      "highlight" : {
        "title" : [ "Introduction to <em>Elasticsearch</em>" ],
        "description" : [ "can meet and greet and I will present on some
➤ <em>Elasticsearch</em> basics and how we use it." ]
      }
      [...]
      "title": "Elasticsearch and Logstash",
      "description": "We can get together and talk about Logstash -
➤ http://logstash.net with a sneak peek at Kibana",
      [...]
      "highlight" : {
        "title" : [ "<em>Elasticsearch</em> and Logstash" ]
      }
    }
  }
]

```

Include which fields you want to highlight.

The reply will contain `_source` as before...

...but also the highlighted fields, if they match the term "elasticsearch".

Highlighting works here because, by default, the title and description fields are included in `_source`. If they had been stored individually (by setting `store` to `true` in the mapping of that field), Elasticsearch would have extracted the contents from the stored field instead of retrieving it from `_source`.

TIP Storing a field and not going through `_source` can be faster if you're highlighting a single field. If you're highlighting multiple fields, using `_source` is typically faster because all fields are fetched in the same trip to the disk. You can force using `_source` even for stored fields by setting `force_source` to `true` in your highlighting request. For most use cases, it's best to stick with the default of using `_source` alone—both in the mapping and for highlighting.

Depending on your use case, the results from listing C.1 might not be what you need. Let's look at two of the most common problems and how you can fix them.

C.1.1 What should be passed on to the user

Results from listing C.1 contain the `_source` field, plus the title and/or description fields if there's something to highlight in them. Assuming you want to return the

title and description fields to the user, you'll have to implement something like this in your application:

- Check if the field (title or description, in this case) is highlighted.
- If it is, show the highlighted fragment. If it's not, take the original field content from `_source`.

A more elegant solution is to have the highlighter return fragments of both the title and the description fields, regardless of whether there's something to highlight in there or not. You'll do that in listing C.2 by setting `no_match_size` to the number of characters you want the fragment to have, if the field doesn't match. The default is 0, which is why fields that don't match don't appear at all.

NOTE Configuring the fragment size is useful when you can't control how large fields are. If you take an event description from `_source` and it fills one page, for example, it will ruin the UI. We'll discuss more about fragment sizes and other fragment options in section C.2.1.

With the highlighter returning all fields you need, the `_source` field from the results becomes redundant, so you can skip returning it by setting `_source` to `false` in your search request, as shown in the next listing.

Listing C.2 Forcing the highlighter to return the needed fields with `no_match_size`

```
curl localhost:9200/get-together/event/_search?pretty -d '{
  "query": {
    "match": {
      "title": "elasticsearch"
    }
  },
  "highlight": {
    "no_match_size": 100,
    "fields": {
      "title": {},
      "description": {}
    }
  },
  "_source": false
}'
# reply
{
  "hits" : [ {
    "_index" : "get-together",
    "_type" : "event",
    "_id" : "103",
    "_score" : 0.9581454,
    "highlight" : {
      "title" : [ "Introduction to <em>Elasticsearch</em>" ],
      "description" : [ "can meet and greet and I will present on some
        em>Elasticsearch</em> basics and how we use it." ]
    }
  }
]
[...]
```

Show up to 100 characters of a field that doesn't match.

You have all the needed information in the highlighted fields, so you disable `_source`.

No `_source` in the results

```

    "highlight" : {
      "title" : [ "<em>Elasticsearch</em> and Logstash" ],
      "description" : [ "We can get together and talk about Logstash -
➡ http://logstash.net with a sneak peek at Kibana" ]

```

This description doesn't match, but the field is shown anyway for completeness.

Highlighting the same fields regardless of whether they match or not is a common use case. Next we'll look at a different (though still common) use case.

C.1.2 *Too many fields contain highlighted terms*

If you pass on the highlighted results of listing C.2 to users, they might get confused by getting `elasticsearch` descriptions highlighted anyway because they searched only in the `title` field. To highlight only fields matching the query, you can set `require_field_match` to `true`, as in the following listing. Now if the query matches the `title` field, only the `title` field gets its terms highlighted.

Listing C.3 Highlighting only fields matching the query

```

curl localhost:9200/get-together/event/_search?pretty -d '{
  "query": {
    "match": {
      "title": "elasticsearch"
    }
  },
  "highlight": {
    "require_field_match": true,
    "fields": {
      "title": {},
      "description": {}
    }
  }
}'
# reply
  "highlight" : {
    "title" : [ "Introduction to <em>Elasticsearch</em>" ]
  }
[...]
```

Only the title field is highlighted now.

Another method to get to the same result is to figure out that the search goes to the `title` field and add only `title` in the list of highlighted fields. This might work, but sometimes you don't have control over which fields are searched on. For example, if you're using the `query_string` query that we discussed in chapter 4, someone could introduce `description:elasticsearch`, even if the default searched field is something else.

`require_field_match` and `no_match_size` are just two of the available highlighting options. There are many more you may find useful, and we'll discuss them in the next sections.

C.2 Highlighting options

Besides choosing which fields to work with, you can configure highlighting with other options, like these:

- Adjusting the size of highlighted fragments and their number
- Changing highlighting tags and encoding
- Specifying a different query for highlighting, instead of the main query

We'll discuss all of these next.

C.2.1 Size, order, and number of fragments

Highlighting `elasticsearch` in an event's description field will show only a fragment of about 100 characters around the highlighted terms. As you might have noticed from listings C.1 and C.2, this doesn't always contain the whole field, so the context could be too large or too small:

```
"description" : [ "can meet and greet and I will present on some
➡ <em>Elasticsearch</em> basics and how we use it." ]
```

We say *about* 100 characters because Elasticsearch tries to make sure that words aren't truncated.

FRAGMENT SIZE

Naturally, there's a `fragment_size` option to change the default fragment size. Setting it to 0 will show the entire field content, which works nicely for short fields like names.

You can set fragment size globally for all fields and individually for each field. Individual settings override global settings, as shown in the next listing, where you'll search for "Elasticsearch," "Logstash," and "Kibana" in the description field.

Listing C.4 Field-specific `fragment_size` setting overrides the global setting

```
curl localhost:9200/get-together/event/_search?pretty -d '{
  "query": {
    "match": {
      "description": "elasticsearch logstash kibana"
    }
  },
  "highlight": {
    "fragment_size": 20,
    "fields": {
      "title": {},
      "description": {
        "fragment_size": "40"
      }
    }
  }
}
```

Global fragment size
applies to all fields

Field-specific fragment
size overrides the
global setting

Fragments
showing
only part
of the field

```

    }
  },
  # reply
  "highlight" : {
    "title" : [ "Logging and <em>Elasticsearch</em>" ],
    "description" : [ "dive for what <em>Elasticsearch</em> is and how
    ➤ it", "logging with <em>Logstash</em> as well as <em>Kibana</em>!" ]
  },
  [...]
  "highlight" : {
    "title" : [ "<em>Elasticsearch</em> and <em>Logstash</em>" ],
    "description" : [ "together and talk about <em>Logstash</em> -
    ➤ http://logstash", "with a sneak peek at <em>Kibana</em>" ]
  },
  [...]
  "highlight" : {
    "title" : [ "<em>Elasticsearch</em> at" ],
    "description" : [ "how they use <em>Elasticsearch</em>" ]
  }
}

```

You can see from this listing that if the fragment size is small enough and there are enough occurrences of the term, multiple fragments are generated.

ORDER OF FRAGMENTS

By default, fragments are returned in the order in which they appear in the text, as you saw in listing C.4. This works well for short texts, where the natural order of fragments gives a better overview of the whole content. For example, the description fragments you got back in listing C.4 do a good job of showing the description.

For large documents, such as books, the natural order doesn't work so well because fragments can be far apart, so the user won't see any link. For example, if you searched for “elasticsearch parent child” in this book, the top two fragments might look like this:

```

"we will discuss how Elasticsearch works and"
"the child aggregation works on buckets generated by"

```

Not terribly relevant, assuming you were looking for parent-child relationships in Elasticsearch. Even though the book itself is relevant because it discusses the topic, it would have been nicer to show a fragment that appears later in the book:

```

"parent-child relationships work with different Elasticsearch documents"

```

When you're highlighting large fields, it makes sense to arrange fragments in the order of their relevance to the query because users are likely to be interested in seeing those relevant parts in order, so they can decide if the result is what they expected.

The highlighter calculates a TF-IDF score for each fragment, much as it calculates scores for documents within the index. To order fragments by this score, you have to set `order` to `score` in the `highlight` part of the request. As is done with fragment sizes, you can set the order individually and/or globally. For example, the following

highlight section will change the order of fragments for the “elasticsearch logstash kibana” query you ran in listing C.4:

```
"highlight": {
  "fields": {
    "description": {
      "fragment_size": 40,
      "order": "score"
    }
  }
}
```

You can see that the fragment matching more terms appears first because it has a higher score:

```
"description" : [ "logging with <em>Logstash</em> as well as
➡ <em>Kibana</em>!", "dive for what <em>Elasticsearch</em> is and how it" ]
```

NUMBER OF FRAGMENTS

With big documents such as books, it makes sense to show only one large, relevant fragment. Multiple small fragments work better for describing smaller fields, like the event descriptions you’ve worked with so far. You can adjust the number of fragments by setting `number_of_fragments` (shocker!), which defaults to 5:

```
"highlight": {
  "fields": {
    "description": {
      "number_of_fragments": 1
    }
  }
}
```

For really small fields, such as names or short descriptions, you can set `number_of_fragments` to 0. This will skip using fragments altogether and return the whole field as a single fragment, ignoring the value of `fragment_size`.

With the size, order, and number of fragments figured out, let’s move on to configuring how those fragments are returned.

C.2.2 Highlighting tags and fragment encoding

You can change the `` and `` tags that are used by default through the `pre_tags` and `post_tags` options. In the following listing, you’ll use `` and `` instead.

Listing C.5 Custom highlighting tags

```
curl localhost:9200/get-together/event/_search?pretty -d '{
  "query": {
    "match": {
      "title": "elasticsearch"
    }
  },
}
```

```

    "highlight": {
      "pre_tags" : ["<b>"],
      "post_tags" : ["</b>"],
      "fields": {
        "title": {}
      }
    }
  },
  # reply
  "highlight" : {
    "title" : [ "<b>Elasticsearch</b> at Rangespan and Exonar" ]
  }
}

```

Global tags; you can also define different tags for each field.

New tags are used in the highlighted fragments.

If your custom tags are HTML like the default ones, you probably want to render the fragments in HTML to show them in some user interface. Here you might encounter a problem: by default, Elasticsearch returns fragments without any encoding, so they won't render properly if there are special characters, such as the ampersand (&). For example, a fragment that's highlighted as `select©` would appear as shown in figure C.2, because the `©` sequence is interpreted as the copyright character.



Figure C.2 The lack of fragment encoding can make the browser interpret HTML incorrectly.

The ampersand needs to be escaped as `&`. You can do that by setting `encoder` to `html`:

```

    "highlight": {
      "encoder": "html",
      "fields": {
        "title": {}
      }
    }
  }
}

```

The HTML encoder will make the text render properly, as shown in figure C.3.

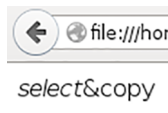


Figure C.3 Using the HTML encoder avoids parsing mistakes.

Now that we've gone through customizing the contents of fragments, let's take a step back and look at the query that generated the highlighted fragments in the first place. By default, terms from the main query are used, but you can define a custom query.

C.2.3 Highlight query

Using the main query for highlighting works for most use cases, but there are some that require special care—for example, if you use rescore queries.

You first met rescoring in chapter 6 when we discussed relevancy, because rescoring allows you to improve the ranking of results by running alternative—often expensive—queries only on the top *N* of the overall result set. Elasticsearch then combines the original score with the score from the rescore queries to get the final ranking. The problem: rescore queries don't apply to highlighting.

This is where custom highlight queries become useful—for example, if the main query is looking for groups with `elasticsearch` or simply `search` in their name, and you also want to boost the presence of tags that end with `search`, like `enterprise search`. A wildcard query for `*search` is expensive, as you saw in chapter 10, section 10.4.1, so you can put this criterion in a rescore query that runs on only the top 200 documents.

In the listing that follows, you'll see how you can put `elasticsearch` and `search` names plus `*search` tags in the highlight query to highlight all the terms involved in the search. You can see that wildcards are expanded and highlight matching tags like `enterprise search`.

Listing C.6 Highlight query contains terms from the main and the rescore query

```
curl localhost:9200/get-together/group/_search?pretty -d '{
  "query" : {
    "match" : {
      "name" : "elasticsearch search"
    }
  },
  "rescore" : {
    "window_size" : 200,
    "query" : {
      "rescore_query" : {
        "wildcard" : {
          "tags.verbatim" : "*search"
        }
      }
    }
  },
  "highlight": {
    "highlight_query": {
      "query_string": {
        "query": "name:elasticsearch name:search tags.verbatim:*search"
      }
    },
    "fields": {
      "name": {},
      "tags.verbatim": {}
    }
  }
}
```

← Main query matches
elasticsearch and search
in the name field

← Rescore query
matches tags
ending in search

← Highlight query
matches all main and
rescore query criteria

```

# reply
  "highlight" : {
    "name" : [ "<em>Elasticsearch</em> Denver" ],
    "tags.verbatim" : [ "<em>elasticsearch</em>" ]
  }
  [...]
  "highlight" : {
    "name" : [ "Enterprise <em>search</em> London get-together" ],
    "tags.verbatim" : [ "<em>enterprise search</em>" ]
  }

```

elastic-search and search are highlighted in the name field.

All tags ending in search are highlighted, too.

Now let's take a deeper look at how highlighting works under the hood. This will allow you to choose the implementation that works best for your use case.

C.3 *Highlighter implementations*

So far we've assumed that you're using the default highlighter implementation called Plain. The Plain Highlighter works by re-analyzing the text from each field to identify terms to highlight and where those terms are located in the text. This is good for most use cases and only requires highlighted fields to be stored, either independently or in the `_source` field. Because it has to analyze the text again, the Plain Highlighter can be slow for large fields; for example, when you index books or blog post contents.

For such use cases, two other implementations come in handy:

- Postings Highlighter
- Fast Vector Highlighter

Both are faster than the Plain Highlighter on large fields, but both require additional data to be stored in the index—data on which their speed is based. Both also come up with their unique features, which will be discussed next.

If it's not obvious which one is best for you, we suggest starting with the Plain Highlighter and moving on to the Postings Highlighter for fields where the Plain Highlighter proves to be too slow, because the Postings Highlighter adds little overhead in terms of index size and also works well if fields are smaller. If the Postings Highlighter doesn't give you the needed functionality, try the Fast Vector Highlighter.

C.3.1 *Postings Highlighter*

The Postings Highlighter requires you to set `index_options` to `offsets` for highlighted fields, which will store each term's location (position and offset) in the index. As you can see in listing C.7, offsets indicate the exact position of a certain term in the text, and with this information, the Postings Highlighter is able to identify which terms to highlight without having to re-analyze the text.

In this listing you'll use the Analyze API, which you first encountered in chapter 5 on analysis.

Listing C.7 Analyze API showing offsets

```

curl localhost:9200/_analyze?pretty -d 'Introduction to Elasticsearch'
# reply
{
  "tokens" : [ {
    "token" : "introduction",
    "start_offset" : 0,
    "end_offset" : 12,
    "type" : "<ALPHANUM>",
    "position" : 1
  }, {
    "token" : "to",
    "start_offset" : 13,
    "end_offset" : 15,
    "type" : "<ALPHANUM>",
    "position" : 2
  }, {
    "token" : "elasticsearch",
    "start_offset" : 16,
    "end_offset" : 29,
    "type" : "<ALPHANUM>",
    "position" : 3
  } ]
}

```

When analyzing text, Elasticsearch can store offsets.

With offsets stored, a second analysis isn't necessary to locate this term.

When analyzing the text, Elasticsearch is able to extract each term's offsets in order to store its exact location. With offsets stored, Elasticsearch doesn't have to analyze the text again during highlighting in order to locate each term. Adding term offsets to the index is a typical tradeoff where you allow slower indexing and a bigger index in order to get better query latency. You saw many such performance tradeoffs in chapter 10.

When you set `index_options` to `offsets`, the Postings Highlighter is used automatically. For example, in the next listing you'll enable offsets for the content field of a new index, add two documents, and highlight them.

Listing C.8 Using the Postings Highlighter

```

INDEX_URL="localhost:9200/test-postings"
curl -XDELETE $INDEX_URL
curl -XPUT $INDEX_URL -d '{
  "mappings": {
    "docs": {
      "properties": {
        "content": {
          "type": "string",
          "index_options": "offsets"
        }
      }
    }
  }
}'

```

Index name for playing with the Postings Highlighter

Required for the Postings Highlighter

Indexing
two sample
documents

```

curl -XPUT $INDEX_URL/docs/1 -d '{
  "content": "Postings Highlighter rocks. It stores offsets in postings."
}'
curl -XPUT $INDEX_URL/docs/2 -d '{
  "content": "Postings are a generic name for the inverted part of the
    ➤ index: term dictionary, term frequencies, term positions."
}'
curl -XPOST $INDEX_URL/_refresh
curl "$INDEX_URL/_search?q=content:postings&pretty" -d '{
  "highlight": {
    "fields": {
      "content": {}
    }
  }
}'
# reply
  "highlight" : {
    "content" : [ "<em>Postings</em> Highlighter rocks.", "It stores
➤ offsets in <em>postings</em>." ]
  }
[...]
  "highlight" : {
    "content" : [ "<em>Postings</em> are a generic name for the inverted
➤ part of the index: term dictionary, term frequencies, term positions." ]
  }

```

Query for postings
in the content
field; Postings
Highlighter is used
automatically.

You can see from this listing that the highlighted samples are sentences, whether large or small. The Postings Highlighter will ignore the `fragment_size` option if you set it; fragments will always be sentences unless you set `number_of_fragments` to 0, in which case the whole field is treated as one fragment.

TIP If you want to set the highlighter implementation manually, you can do so by setting `type` to `plain` (for the Plain Highlighter), `postings` (for the Postings Highlighter), or `fvh` (for the Fast Vector Highlighter). This can be done globally or per field and is useful if you change your mind about the implementation and you don't want to re-index. For example, you index offsets but don't like the sentence-as-fragment approach of the Postings Highlighter, so you need a way to get back to using the Plain Highlighter.

Internally, the Postings Highlighter breaks the field into sentences (which then become fragments) and treats those sentences as separate documents, scoring them by using BM25 similarity. As we discussed in chapter 6, BM25 is a TF-IDF-based similarity that works well for short fields, like your sentences are supposed to be.

Because of the way it creates and scores fragments, the Postings Highlighter works well when you're indexing natural language, such as books or blogs. It might not work so well when you're indexing code, for example, because the concept of a sentence often doesn't work, and you can end up with the entire field as a single fragment and no options to reduce the fragment size.

Another downside of the Postings Highlighter is that, at least in version 1.4, it doesn't work well with phrase queries because it only accounts for individual terms.

For example, in the next listing you'll look for the phrase "Elasticsearch intro" by using a `match_phrase` query.

Listing C.9 Postings Highlighter matches all the terms and discounts phrases

```
curl -XPUT localhost:9200/test-postings/docs/2 -d '{
  "content": "Elasticsearch intro - first you get an intro of the core
  ➤ concepts, then we move on to the advanced stuff"
}'
curl localhost:9200/test-postings/_search?pretty -d '{
  "query": {
    "match_phrase": {
      "content": "Elasticsearch intro"
    }
  },
  "highlight": {
    "encoder": "html",
    "fields": {
      "content": {}
    }
  }
}'
# reply
"highlight": {
  "content": ["<em>Elasticsearch</em> <em>intro</em> - first you get an
  ➤ <em>intro</em> of the core concepts, then we move on to the advanced
    stuff"]
}
curl localhost:9200/test-postings/_search?pretty -d '{
  "query": {
    "match_phrase": {
      "content": "Elasticsearch intro"
    }
  },
  "highlight": {
    "encoder": "html",
    "fields": {
      "content": {
        "type": "plain"
      }
    }
  }
}'
#reply
"highlight" : {
  "content" : [ "<em>Elasticsearch</em> <em>intro</em> - first you get an
  ➤ intro of the core concepts, then we move on to the advanced stuff" ]
}
```

Second occurrence of
intro is highlighted,
even though it's not
part of the phrase

With the Plain
Highlighter, only the
phrase is highlighted.

You get individual terms highlighted even if they don't belong to the phrase, which doesn't happen with the Plain Highlighter. On the upside, although indexing offsets increase your index size and slow down indexing a bit, the overhead is lower than what you get when adding term vectors, which are needed by the Fast Vector Highlighter.

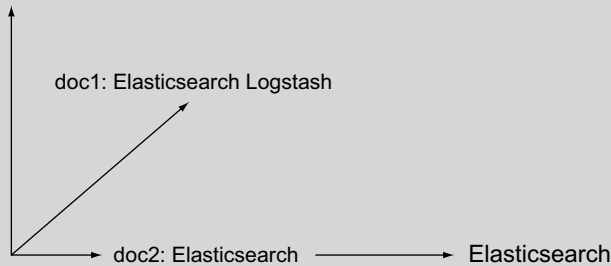
C.3.2 Fast Vector Highlighter

To enable the Fast Vector Highlighter for a field, you have to set `term_vector` to `with_positions_offsets` in the mapping. This will allow Elasticsearch to identify terms as well as their location in the text without re-analyzing the field content. For large fields—for example, those over 1 MB—the Fast Vector Highlighter is faster than the Plain Highlighter.

What are term vectors?

Term vectors are a way to represent documents by using terms as dimensions. For example, the following diagram represents a document with the `Elasticsearch` and `Logstash` terms and another document containing only `Elasticsearch`.

Logstash



Metadata, vectors, and rankings

You can also represent a query as another vector and rank documents based on the distance between the query vector and each document's vector. Another application is to add other metadata to each document—for example, the field's total size—that will influence ranking. For more information about term vectors and their use, go to https://en.wikipedia.org/wiki/Vector_space_model.

For highlighting, this metadata has to be the list of positions and offsets for each term. This is why the Fast Vector Highlighter needs the `with_positions_offsets` setting. Alternative settings are `no` (default), `yes`, `with_positions`, and `with_offsets`.

Compared to the Postings Highlighter, the Fast Vector Highlighter takes up more space and requires more computation during indexing, because both need positions and offsets, but only the Fast Vector Highlighter has to compute the term vectors themselves, which are disabled by default.

When `term_vector` is set to `with_positions_offsets` for a field, Elasticsearch automatically uses the Fast Vector Highlighter for that field. For example, the get-together

event and group descriptions from the code samples use this highlighter by default. Here's a relevant snippet from the mapping:

```
"group" : {
  "properties" : {
    "description" : {
      "type" : "string",
      "term_vector": "with_positions_offsets"
```

Compared to the Postings Highlighter, this offers better phrase highlighting. Instead of highlighting every matching term, the Fast Vector Highlighter highlights only terms belonging to the phrase—as the Plain Highlighter did in listing C.9.

The Fast Vector Highlighter also comes with unique functionality:

- It works nicely with multi-fields, because it's able to combine matches from multi-fields into the same set of fragments,
- If there are multiple words to highlight, you can highlight them with different tags.
- You can configure how the boundaries of a fragment are selected.

Let's take a deeper look at each of these features.

HIGHLIGHTING MULTI-FIELDS

You met multi-fields in chapter 3, section 3.3.2, as a way to index the same text in multiple ways. Multi-fields are a great way to refine your searches, but highlighting them properly may be tricky if variations of the same field produce different matches. Take the following listing, for example, where the `description` field is analyzed in two ways: the default is the `english` analyzer, which uses stemming to match search with searching. The `suffix` subfield uses a custom analyzer that makes use of Edge ngrams to match words with common suffixes, such as `elasticsearch` and `search`. When you do a `multi_match` query on both of them, the Plain Highlighter can match only one field at a time.

Listing C.10 Plain Highlighter doesn't work well with multi-fields

```
curl -XPUT localhost:9200/multi -d '{
  "settings": {
    "analysis": {
      "analyzer": {
        "my-suffix": {
          "tokenizer": "standard",
          "filter": ["lowercase", "suffix"]
        }
      },
      "filter": {
        "suffix": {
          "type": "edgeNGram",
          "min_gram": 5,
          "max_gram": 5,
```

Custom analyzer that accounts for only the last five letters of each term

```

        "side": "back"
      }
    }
  },
  "mappings": {
    "event": {
      "properties": {
        "description": {
          "type": "string",
          "analyzer": "english",
          "term_vector": "with_positions_offsets",
          "fields": {
            "suffix": {
              "type": "string",
              "analyzer": "my-suffix",
              "term_vector": "with_positions_offsets"
            }
          }
        }
      }
    }
  }
},
curl -XPUT localhost:9200/multi/event/1 -d '{
  "description": "elasticsearch is about searching"
}',
curl localhost:9200/multi/_refresh
curl -XGET localhost:9200/multi/event/_search -d'
{
  "query": {
    "multi_match": {
      "query": "search",
      "fields": ["description", "description.suffix"]
    }
  },
  "highlight": {
    "type": "plain",
    "fields": {
      "description": {},
      "description.suffix": {}
    }
  }
},
# reply
"highlight": {
  "description": ["elasticsearch is about <em>searching</em>"],
  "description.suffix": ["<em>elasticsearch</em> is about searching"]
}

```

English analyzer does stemming on the default field, matching search with searching

Custom analyzer takes suffixes only, matching elasticsearch with search

Plain Highlighter can highlight only one match or the other.

Here's where the Fast Vector Highlighter comes to the rescue because it can combine both multi-fields into one and highlight all the matches. It only requires `term_vector` to be set to `with_positions_offsets` on all the fields you need to highlight (which is the requirement for the Fast Vector Highlighter to work in the first place). You

already added this in this listing. To combine multiple subfields into one, you have to indicate which subfields you want to highlight with the `matched_fields` option:

```
"highlight": {
  "fields": {
    "description": {
      "matched_fields": ["description", "description.suffix"]
    }
  }
}
```

With the document and the query from listing C.10, you'll have the highlighting that you'd expect:

```
"highlight": {
  "description": ["<em>elasticsearch</em> is about <em>searching</em>"]
}
```

USING DIFFERENT TAGS FOR DIFFERENT FRAGMENTS

To **bold** the first highlighted word and *italicize* the second, you can specify an array of tags:

```
"highlight": {
  "fields": {
    "description": {
      "pre_tags": ["<b>", "<em>"],
      "post_tags": ["</b>", "</em>"]
    }
  }
}
```

If there are more than two words to highlight, the Fast Vector Highlighter starts over: bold the third, italicize the fourth, and so on. If you have many words to highlight, you might want to keep track of their number. You can do that by setting `tags_schema` to `styled`, like in this query:

```
"query": {
  "match": {
    "description": "elasticsearch logstash kibana"
  }
},
"highlight": {
  "tags_schema": "styled",
  "fields": {
    "description": {}
  }
}
```

If you run it on the documents from the code samples, you'll get the first hit highlighted like this:

```
      "highlight": {
        "description": [
          "for what <em class=\"h1t1\">Elasticsearch</em> is and how
➡ it can be used for logging with <em class=\"h1t2\">Logstash</em> as well
➡ as <em class=\"h1t3\">Kibana</em>!"
        ]
      }
```

This allows you to take the class name (`h1tX`) and figure out which words matched first, second, and so on.

CONFIGURING BOUNDARY CHARACTERS

Recall from section C.2.1 that we said `fragment_size` is approximate because Elasticsearch tries to make sure words aren't truncated. If you thought then that the explanation is a bit vague, it's because the behavior depends on the highlighter implementation.

With the Postings Highlighter, fragment size is irrelevant because it breaks the text down into sentences. The Plain Highlighter adds terms around the highlighted term until it gets close to the fragment size, which means the boundary is always a term. As you've seen in the listings of this chapter, this works well for natural language, but it might become problematic in other use cases where the word and term concepts don't overlap. For example, if you're indexing code, you may have variable definitions like this:

```
variable_with_a_very_very_very_very_long_name = 1
```

To search this kind of text effectively, you'll need an analyzer that can break this long variable and allow you to search for terms within it.

TIP You can do this with the Pattern Tokenizer, where you specify a pattern that includes underscores—for example, `(\\ |_)`—which will tokenize on spaces and underscores. In chapter 5 you'll find more information about analyzers and tokenizers.

If the analyzer will break the variable into tokens, the Plain Highlighter will break it, too, even if you don't want it to. For example, a search for `long` with a fragment size of 20 would give you this:

```
_very_very_very_very_<em>long</em>_name = 1
```

The Fast Vector Highlighter works differently because words aren't the same as terms. Words are strings delimited by the following characters: `. , ! ? \t \n`. You can change the list through the `boundary_chars` option. When it builds fragments, it seeks those characters for `boundary_max_scan` characters (defaults to 20) from the limits that are normally set by `fragment_size`. If it doesn't find such boundary characters while scanning, the fragment is truncated. By default, the Fast Vector Highlighter will truncate the code sample while highlighting `long`:

```
ry_very_<em>long</em>_name = 1
```

You can fix this by changing the defaults in two ways. One is to add the underscore to the list of boundary characters. This will still truncate the variable but in a more predictable way:

```
"highlight": {
  "fields": {
    "description": {
```

```

        "fragment_size": 20,
        "boundary_chars": ". , ! ? \t \n _"
# will yield
very_very_<em>long</em>_name = 1

```

The other option is to leave `boundary_chars` set to the default and extend `boundary_max_scan` instead, which will increase the chances of having the whole variable included in the fragment, even if it implies a higher fragment size for this particular fragment:

```
variable_with_a_very_very_very_very_<em>long</em>_name = 1
```

Issues with fragment boundaries are typically visible when you need small fragments. For bigger chunks, inaccurate boundaries are less likely to be visible to users because their attention tends to focus on the highlighted bits and the words around them, not on the fragment as a whole. Another parameter to configure for the Fast Vector Highlighter is the `fragment_offset`. With this parameter you can control the margin to start the highlighting from.

LIMITING THE NUMBER OF MATCHES FOR THE FAST VECTOR HIGHLIGHTER

The final configuration option we discuss is the `phrase_limit` parameter. If the Fast Vector Highlighter matches many phrases, it could consume a lot of memory. By default, only the 256 first matches are used. You can change this amount using the `phrase_limit` parameter.

Elasticsearch IN ACTION

Gheorghe • Hinman • Russo



Modern search seems like magic—you type a few words and the search engine appears to know what you want. With the Elasticsearch real-time search and analytics engine, you can give your users this magical experience without having to do complex low-level programming or understand advanced data science algorithms. You just install it, tweak it, and get on with your work.

Elasticsearch in Action teaches you how to write applications that deliver professional quality search. As you read, you'll learn to add basic search features to any application, enhance search results with predictive analysis and relevancy ranking, and use saved data from prior searches to give users a custom experience. This practical book focuses on Elasticsearch's REST API via HTTP. Code snippets are written mostly in bash using cURL, so they're easily translatable to other languages.

What's Inside

- What is a great search application?
- Building scalable search solutions
- Using Elasticsearch with any language
- Configuration and tuning

Perfect for developers and administrators building and managing search-oriented applications.

Radu Gheorghe is a search consultant and software engineer. **Matthew Lee Hinman** develops highly available, cloud-based systems. **Roy Russo** is a specialist in predictive analytics.

Technical editor: **Jetro Coenradie**

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/elasticsearch-in-action

“To understand how a modern search infrastructure works is a daunting task. Radu, Matt, and Roy make it an engaging, hands-on experience.”
—Sen Xu, Twitter Inc.

“An indispensable guide to the challenges of search of semi-structured data.”
—Artur Nowak, Evidence Prime

“The best resource for a complex topic. Highly recommended.”
—Daniel Beck, juris GmbH

“Took me from *confused* to *confident* in a week.”
—Alan McCann, Givsum.com

