Elasticsearch Grails Plugin - Samples

Puneet Behl

Version 0.1

Table of Contents

1. Installation 1
2. Boost 2
    2.1. Index time `boost` 'or' Static boosting 2
    2.2. Query time `boost` 'or' Dynamic boosting 4

Chapter 1. Installation

Chapter 2. Boost

If you don't know what does `boost` means in Elasticsearch. I would suggest to give a quick read at Elasticsearch documentation on `boost`

In Grails Elasticsearch plugin, `boost` is possible in two ways:

- Index time boost or Static boosting
- Query time boost

2.1. Index time `boost` 'or' Static boosting

In this `boost` factor is calculated based on the nature of document.

> Using index time boosting you need to reindex to change the boosting behaviour.

The `boost` factor at index time sets a document related boost. That is, a factor depending on the nature of the document is being set in the relevance scoring formula.

2.1.1. Example:

The number of links pointing to this document (also known as popularity or authority ranking in web search). Such a boost factor is also called "static boost" since it does not change from query to query. If static boost must change, the whole document must be reindexed.

```
PUT esdemo_v0
{
  "mappings": {
    "book": {
      "properties": {
        "title": {
          "type": "string",
          "boost": 2
        },
        "content": {
          "type": "string"
        }
      }
    }
  }
}
```

> Matches on the title field will have twice the weight as those on the content field, which has the default boost of 1.0.

The only advantage that index time boosting has is that it is copied with the value into the `_all` field. This means that, when querying the `_all` field, words that originated from the title field will have a higher score than words that originated in the content field. This functionality comes at a cost: queries on the _all field are slower when index-time boosting is used.

You can achieve this using Elasticsearch Grails plugin as:

```
package esdemo

class Book {
    String title
    String content
    String datePublished

    static searchable = {
        title boost: 2.0
        content boost: 1.0
    }
}
```

Now, when you query on `Book` domain, then you wiil notice that in result, instance having "Elasticsearch" in `title` field has double score as compared to other having "Elasticsearch" in content. Following is the test case which illustrate this:

```
    void "should double the score of search hit when 'Elasticsearch` is found in title
field as compared to content field"() {
        setup:
        List<Book> books =[]
        books << new Book(title: "Elasticsearch Introduction", content: "basic
introduction", datePublished: new Date()).save(flush: true, failOnError: true)
        books << new Book(title: "Hello World", content: "Elasticsearch",
datePublished: new Date()).save(flush: true, failOnError: true)

        when:
        elasticSearchService.index(Book)
        elasticSearchAdminService.refresh()
        def results = Book.search("Elasticsearch", [score: true])

        then:
        results.scores["1"] == 2 * results.scores["2"]

        cleanup:
        elasticSearchService.unindex(Book)
        Book.deleteAll()
    }
```

Result

```
results.scores["1"] == 2 * results.scores["2"]
|       |     |     |    | |       |     |
|       |     |     true | |       |     |
|       |     0.2972674  | |       |     0.1486337
|       |                | |       [1:0.2972674, 2:0.1486337]
|       |                | [total:2, searchResults:[esdemo.Book : 1, esdemo.Book : 2],
scores:[1:0.2972674, 2:0.1486337]]
|       |                0.2972674
|       [1:0.2972674, 2:0.1486337]
[total:2, searchResults:[esdemo.Book : 1, esdemo.Book : 2], scores:[1:0.2972674,
2:0.1486337]]
```

2.2. Query time `boost` 'or' Dynamic boosting

We can control the relative weight of any query clause by specifying a boost value, which defaults to 1. A boost value greater than 1 increases the relative weight of that clause. So we could rewrite the preceding query as follows:

```
GET /_search
{
    "query": {
        "bool": {
            "must": {
                "match": {
                    "content": {
                        "query":    "full text search",
                        "operator": "and"
                    }
                }
            },
            "should": [
                { "match": {
                    "content": {
                        "query": "Elasticsearch",
                        "boost": 3
                    }
                }},
                { "match": {
                    "content": {
                        "query": "Lucene",
                        "boost": 2
                    }
                }}
            ]
        }
    }
}
```

The `boost` factor given at query time is query term related. That is, a factor depending on a term in the query is being set in the relevance scoring formula. Example: hits on docs with a certain term should be ranked higher than hits on docs with other terms in the query. Such a boost factor is called "dynamic boost" because it can change from query to query. There is no need to reindex docs.

Query term boost can be done in Elasticsearch Grails plugin as:

```
    void "dynamic boosting, hits on docs with a certain term should be ranked higher
than hits on docs with other terms in the query"() {
        setup:
        List<Book> books =[]
        books << new Book(title: "Elasticsearch Introduction", content: "basic
introduction", datePublished: new Date()).save(flush: true, failOnError: true)
        books << new Book(title: "Hello Lucene", content: "Elasticsearch",
datePublished: new Date()).save(flush: true, failOnError: true)
        books << new Book(title: "Hello World", content: "Elasticsearch",
datePublished: new Date()).save(flush: true, failOnError: true)

        when:
        elasticSearchService.index(Book)
        elasticSearchAdminService.refresh()
        Closure query = {
            bool {
                should {
                    match {
                        title(query: "Elasticsearch", boost: 3.0)
                    }
                }
                should {
                    match {
                        title(query: "Lucene", boost: 2.0)
                    }
                }
            }
        }
        def results = Book.search (query, null, [score: true])

        then:
        (2 * results.scores["3"])?.round(6) == (3 * results.scores["4"])?.round(6)

        cleanup:
        elasticSearchService.unindex(Book)
        Book.deleteAll()
    }
```

Result

```
2 * results.scores["3"] == 3 * results.scores["4"]
  | |       |     |     |    | |       |     |
  | |       |     |     true | |       |     0.48725736
  | |       |     0.730886   | |       [3:0.730886, 4:0.48725736]
  | |       |                | [total:2, searchResults:[esdemo.Book : 3, esdemo.Book :
4], scores:[3:0.730886, 4:0.48725736]]
  | |       |                1.461772
  | |       [3:0.730886, 4:0.48725736]
  | [total:2, searchResults:[esdemo.Book : 3, esdemo.Book : 4], scores:[3:0.730886,
4:0.48725736]]
  1.461772
```

In the above query, please note that:

- "Elasticserach" found in title is given 3 times more importance
- "Lucene" found in title is given 2 times more importance