How scoring works in Elasticsearch

Published Feb 18, 2016

#RELEVANCE SCORING #ELASTICSEARCH

In this article, we'll take a look at how relevancy scoring is done in Elasticsearch, touching on information retrieval concepts and the mechanisms used to determine the relevancy score of a document for a given query. We'll also point out some "gotchas" and common confusion points along the way.

Introduction

Relevance, like beauty, is in the eye of the beholder. Search algorithms try to bring some empiricism to this area by employing models, rules and mathematical calculations to return and appropriately rank the results that most people would expect. They're usually pretty good at that because the field of Information Retrieval is continually maturing and those algorithms are getting more and more sophisticated every day. However, because relevance is subjective, there is no way to return the perfect result set. There are, however, various approaches and tools that can be used to tune the result set for the most optimal results for your users.

In our next article, we'll discuss strategies and techniques to take advantage of the built-in tools Elasticsearch provides that we can use to manipulate relevancy scores, but first we need to have a solid understanding of how those scores are determined before we start fiddling with the knobs and turning the dials.

Scoring Basics

Before Elasticsearch starts scoring documents, it first reduces the candidate documents down by applying a boolean test - does the document match the query? Once the results that match are retrieved, the score they receive will determine how they are rank ordered for relevancy.

The scoring of a document is determined based on the field matches from the query specified and any additional configurations you apply to the search. We'll get into scoring details in just a minute, but first, be aware that just because there is a match does not mean the document is relevant to your users. For example, a user searching on "apple" could mean the company or the fruit, but matches may occur on documents for both the company and for the fruit. Some ways to handle this situation include filtering matches by index, by document type (or other facets), or by applying some contextual or personalized logic, but the point we're trying to make here is that just getting a match to one or more terms in a document field does not equate with relevance. Likewise, just because we didn't get a match, doesn't mean the document isn't relevant.

The Practical Scoring Function

Elasticsearch runs Lucene under the hood so by default it uses Lucene's Practical Scoring Function. This is a similarity model based on Term Frequency (tf) and Inverse Document Frequency (idf) that also uses the Vector Space Model (vsm) for multi-term queries. If all that jargon makes you feel lost already, don't worry. It's all handled for you behind the scenes so that you need to have only a basic understanding of the models to follow along. This article is here to help with that. If you want to know even more about scoring from the official source, check out Lucene's documentation on scoring.

Let's start with a simple overview of the default formula from the *Elasticsearch - The Definitive Guide* section on relevance. It shows us which mechanisms are at play in determining relevancy:

score(q,d) =
queryNorm(q)
* coord(q,d)
* SUM (
tf(t in d),
idf(t)²,
t.getBoost(),
norm(t,d)
) (t in q)

- score(q,d) is the relevance score of document d for query q.
- queryNorm(q) is the query normalization factor.
- coord(q,d) is the coordination factor.
- The sum of the weights for each term t in the query q for document d.
  - tf(t in d) is the term frequency for term t in document d.
  - idf(t) is the inverse document frequency for term t.
  - t.getBoost() is the boost that has been applied to the query.
  - norm(t,d) is the field-length norm, combined with the index-time field-level boost, if any.

Now, let's get more familiar with each of the scoring mechanisms that make up the Practical Scoring Function:

- **Term frequency (tf)**: This is the square root of the number of times the term appears in the field of a document:

```
tf = sqrt(termFreq)
```

Term frequency clearly assumes that the more times a term appears in a document, the higher its relevancy should be. Usually that's the case and you'll probably continue to use this scoring mechanism, but if you just need to know that the term appears in the document at all and you don't care how many times, you can configure the field to ignore term frequency during indexing. A better way to handle that situation, though, is to apply a filter using the term at query time. Note, too, that inverse document frequency can't be turned off so, even if you disable term frequency, the inverse document frequency will still play a role in the scoring. Finally, note that not-analyzed fields (typically those where you expect an exact match) will automatically have term frequency turned off.

- **Inverse document frequency (idf)**: This is one plus the natural log (as in "logarithm", not "log file") of the documents in the index divided by the number of documents that contain the term:

```
idf = 1 + ln(maxDocs/(docFreq + 1))
```

What inverse document frequency captures is that, if many documents in the index have the term, then the term is actually less important than another term would be where few documents include the term.

- **Coordination (coord)**: Counts the number of terms from the query that appear in the document.

With the coordination mechanism, if we have a 3-term query and a document contains 2 of those terms, then it will be scored higher than a document that has only 1 of those terms. Like term frequency, coordination can be turned off, but that is typically only done when the terms are synonymous with each other (and therefore, having more than one of them does not increase relevancy). A better way to handle that situation, though, is to populate a synonym file to handle synonyms automatically.

- **Field length normalization (norm)**: This is the inverse square root of the number of terms in the field:

```
norm = 1/sqrt(numFieldTerms)
```

For field length normailization, a term match found in a field with a low number of total terms is going to be more important than a match found in a field with a large number of terms. As with term frequency and coordination, you can choose to not implement field length norms in a document (the setting applies to all fields in the document). While you can save memory by turning this off, you may lose some valuable scoring input. The only time it might make sense to turn off this feature is similar to the case for turning off term frequency - when it does not matter how many terms there are, but only that the query term exists. Still, there are other ways of handling such a situation (like using a filter instead). Note that not-analyzed fields will have field length normalization disabled by default.

- **Query normalization (queryNorm)**: This is typically the sum of squared weights for the terms in the query.

Query normalization is used so that different queries can be compared. For any individual query, however, it uses the same score for every document (effectively negating its impact within an individual query) so it's not something we need to spend any time on.

- **Index boost**: This is a percentage or absolute number used to boost any field at index time.

Note that in practice an index boost is combined with the field length normalization so that only a single number will be stored for both in the index; however, Elasticsearch strongly recommends against using index-level boosts since there are many adverse effects associated with this mechanism.

- **Query boost**: This is a percentage or absolute number that can be used to boost any query clause at query time.

Query boosting allows us to indicate that some part(s) of the query should be more important than other parts. Documents will be scored accordingly to their matches for each part. It can also be used to boost a particular index if you're searching across multiple indexes and want one to have more importance. There are quite a few options that can be used to boost a score at query time, but we'll have to save those details for our next article since it's too much to cover here. Note that in practice, these boosts are combined with the queryNorm when applying `explain` (which we'll look at below) so you will see queryNorms of different values if you have used a boost at query time and performed an `explain`.

Note that term frequency, inverse document frequency, and field-length normalization are stored for each document at index time. These are used to determine the weight of a term in a document.

Explain

Let's bring in an example so you can see how the Practical Scoring Function formula is applied.

In our Elasticsearch instance, we've indexed the top 250 films according to IMDB voters. Let's take a look at how relevancy scores are determined by using the Explain API. Get ready to do some math!

`explain` requires the index name (in our case that's "top_films"), the document type (for us that's "film"), and the id number of a specific document (here we're using id 172... Monty Python's "Life of Brian"). We're running a simple match query in the title field on the term "life":

```
curl -XGET 'https://aws-us-east-1-portal10.dblayer.com:10019/top_f
{
   "query" : {
      "match" : {
         "title" : "life"
      }
   }
}
```

Here's what `explain` returns to us about the match and the score:

```
{
   "_type" : "film",
   "_index" : "top_films",
   "_id" : "172",
   "matched" : true,
   "explanation" : {
      "description" : "weight(title:life in 38) [PerFieldSimilari
      "value" : 1.9067053,
      "details" : [
         {
            "details" : [
               {
                  "description" : "queryWeight, product of:",
                  "value" : 0.999999940000001,
                  "details" : [
                     {
                        "description" : "idf(docFreq=2, maxDocs=5
                        "value" : 3.8134108
                     },
                     {
                        "value" : 0.26223242,
                        "description" : "queryNorm"
                     }
                  ]
               },
               {
```

```
                "description" : "fieldWeight in 38, product of:
                "value" : 1.9067054,
                "details" : [
                   {
                      "description" : "tf(freq=1.0), with freq
                      "details" : [
                         {
                            "value" : 1,
                            "description" : "termFreq=1.0"
                         }
                      ],
                      "value" : 1
                   },
                   {
                      "value" : 3.8134108,
                      "description" : "idf(docFreq=2, maxDocs=5
                   },
                   {
                      "value" : 0.5,
                      "description" : "fieldNorm(doc=38)"
                   }
                ]
             }
          ],
          "value" : 1.9067053,
          "description" : "score(doc=38,freq=1.0), product of:"
       }
    ]
 }
}
```


First, what we see is that the results confirm the index name, document type, and document id that we requested. Next, we see "matched" is true. This is the boolean part of the function - the document either matches or it doesn't. Remember that scoring is only performed for documents that match. Because we have a match, we then have a detailed explanation of the relevancy score and the value of the final score. Let's take this piece by piece.

The first "description" element is just a shorthand overview for how the score was computed (the "38" you see there is just an internal document identifier - it doesn't actually mean anything about the calculation). What is most important is the final relevancy score determined for this document for our query, which was 1.9067053. The "details" section tells us how the score was calculated and, as you can see, contains details within details for sub-calculations. We actually have two sets of details - one for the query weight and one for the field weight - before the final score was arrived at.

Let's focus on the field weight details since that's where we can make an impact once we start manipulating the score with the built-in tools.

Deconstructing Field Weight

First we see the term frequency, which has a value of 1. That's because in the title "Life of Brian", the term "life" occurs only once and the square root of 1 is 1.

Next, we see the inverse document frequency with a value of 3.8134108 using "docFreq=2" and "maxDocs=50". That's calculated as:

```
1 + ln(maxDocs/(docFreq + 1))
```

So, if we plug in the numbers, we get:

```
1 + ln(50/(2 + 1)) = 3.8134108
```

If you're scratching your head because we have 50 as our maxDocs number, but you know we said we indexed 250 top films, hold that thought! You're on to something important and we'll cover it in the next section.

Finally, we see the field length normalization. It has a value of 0.5. That's calculated as:

```
1/sqrt(numFieldTerms)
```

In our case, because there are 3 terms in the title "Life of Brian", that's...

```
1/sqrt(3) = 0.57735
```

However, because norms are stored as a single byte in Elasticsearch, our field length norm gets truncated to just 0.5 and we lose the decimal precision. Note here, the qualifier "(doc=38)". That's just the internal document id for this request, which we also mentioned appears in the "description". It has nothing to do with the calculation other than being a reference to this particular document.

Reconstructing Field Weight

So, if we multiply those three measures together (tf * idf * norm), we get a score of 1.9067054 for the field weight. If we then multiply that by the score determined in the query weight section (0.999999940000001), which is used to determine the relative importance of our query compared with other queries, we get the final score of 1.9067053.

What we don't see here from the Practical Scoring Function formula is our coordination factor. That's because it is a 1. We only searched for one term and it was found so the coordination calculation is not going to impact the final score of this document. It'd just be multiplied together with the field weight and the query weight. Our final score would be the same.

Additionally, we did not apply any index or query boosts since we wanted to show the default scoring behavior. We'll get into how and when to set boosts in our next article.

The Sharding Effect

Now, remember that 50 maxDocs in the inverse document frequency calculation from the example above? Why 50 instead of the 250 films we have indexed? As you may have guessed from the title of this section, it's because of sharding.

Compose Elasticsearch deployments include 5 shards automatically. When we indexed our documents, we didn't make any specification about how sharding should be applied so the documents got doled out evenly across each of the shards - 50 documents on each of our 5 shards = 250 documents.

So then, when our query found a match to our document, it counted the number of documents found on that particular shard for use in the inverse document frequency calculation.

Another concern caused by this behavior is with the docFreq (the total number of documents which had a match). Actually there were 3 documents across the index that should have matched our query, not 2. The problem is that only 2 matches were found in the particular shard where "Life of Brian" was stored. The third match was located in another shard so it

wasn't identified. Besides the 50 maxDocs being inaccurate, the docFreq of 2 was also inaccurate. It should have been 3. Yeah... Gotcha!

You can see how the sharding effect could significantly impact the relevancy scores of your result set. All else being equal, a document found on a shard with more total documents would be scored lower than a document on a shard with less total documents. A document found on a shard with more additional matching documents would be scored lower than one found on a shard with lower or no additional matching documents. Not so good.

How do you combat the sharding effect? There are a couple different ways.

- **Document routing**: You can use document routing to make sure documents from a single index all go to the same shard by using the value of a specified field. This assumes that your searches will be performed against a single index or on multiple indexes that live on the same shard. You'll want to use the routing field in your search request as well as at index time.

- **Search type**: Search type lets you specify an order of events you want the search to perform. For this situation the "dfs_query_then_fetch" will solve our problem. It will query all the shards to get the frequencies distributed across them, then perform the calculations on the matching documents.

Using Search Type

Since we weren't keen on reindexing our documents, we opted for the search type solution. Unfortunately, it can't be used directly with the Explain API, but what we can do is use the Search API to perform a search using the "dfs_query_then_fetch" search type and add a parameter of "explain=true" to get the scoring explanation. Here's our search request:

```
curl -XGET 'https://aws-us-east-1-portal7.dblayer.com:10304/top_fi
{
   "query" : {
      "match" : {
         "title" : "life"
      }
   }
}
```

Note that because we are doing a full-fledged search here, we don't need to specify a document id that we are interested in. Instead, all matching results will be returned with the details of their scoring explanations. While using "explain=true" in search is a great tool for tuning search results for optimal relevancy, make sure not to leave it set in your production queries since that would be a very performance-expensive call to make for each search.

We actually retrieved 3 results with this search (as mentioned above), but let's just look at part of the result for "Life of Brian" so that we can compare it with what we saw above:

```
{
"_index" : "top_films",
"_node" : "IpZTwukkTFuz2_yHnEFMpw",
"_type" : "film",
"_score" : 2.5675833,
"_shard" : 3,
"_id" : 172,
"_explanation" : {
   "details" : [
      {
         "details" : [
            {
               "details" : [
                  {
                     "value" : 1,
                     "description" : "termFreq=1.0"
                  }
               ],
               "description" : "tf(freq=1.0), with freq of:",
               "value" : 1
            },
            {
               "value" : 5.1351666,
               "description" : "idf(docFreq=3, maxDocs=250)"
            },
            {
               "value" : 0.5,
               "description" : "fieldNorm(doc=40)"
            }
         ],
         "value" : 2.5675833,
         "description" : "fieldWeight in 40, product of:"
      }
   ],
   "value" : 2.5675833,
   "description" : "weight(title:life in 40) [PerFieldSimilarit
}
```


First of all, you'll notice the node, shard, and score are retrieved along with the explanation. The document source is also retrieved with search results, but we did not display it here for simplicity's sake.

So here, the internal document id for this request is 40. Again, not something that figures in the calculation, but we just want to clarify that's what you're seeing where it shows "(doc=40)" and where it's mentioned in the description. Our term frequency still has the value of 1 and our field length normalization still has the value of 0.5 since neither of those calculations changed. Where we do see a difference is with the inverse document frequency. Now that we're comparing across all 250 documents, the score is much higher since very few documents actually contain the term "life" in the title. Only 3, in fact. And knowing there are 3 matches, not 2 or 1, also makes a difference in the score.

The inverse document frequency score for this document now gets calculated like this:

```
1 + ln(250/(3 + 1)) = 5.1351666
```

That's a big difference!

Setting a Search Preference

Finally, you may be interested in one other helpful search setting:

- **Search preference**: You can use search preference to specify the nodes, shards, primary, or replicas you want the search to apply to. While this does not solve the sharding effect problem, it is included here so that, depending on how you index to your shards and configure your replicas, you know you can control precisely where your searches are being performed.

Wrapping Up

We hope that reviewing some core concepts and walking through a simple example in this article has helped clarify how the default scoring works in Elasticsearch. We've also tried to indicate some considerations for you to keep in mind and to warn you of some things to watch out for so that you can configure your Elasticsearch for optimal results. Next we'll start turning the dials and fiddling with the knobs...

Lisa Smith - keepin' it simple. Love this article? Head over to Lisa Smith's author page to keep reading.

RELATED ARTICLES

MAR 31, 2016

Elasticsearch Query-Time Strategies and Techniques for Relevance: Part II

In this article, we're going to look at some of the built-in tools that Elasticsearch provides for impacting relevance scores...

Lisa Smith

MAR 24, 2016

Elasticsearch Query-Time Strategies and Techniques for Relevance: Part I

In this 2-part series, we'll look at a couple different strategies for applying some of Elasticsearch's built-in tools at que...

Lisa Smith

JAN 13, 2020

Compose Makes Elasticsearch Major Version Upgrades Easier

Compose for Elasticsearch users now have the ability to upgrade to a new major version with a click of a button right from th...

Abdullah Alger