> 参考文章：https://www.cnblogs.com/sheeva/p/6847309.html

Elasticsearch提供了以下相似度模型：默认的tf-idf模型、bm25、drf和ib。

tf-idf 模型

Elasticsearch中的默认相似度模型是tf/idf模型。tf/idf模型是最常见的向量空间模型(vsm)。在向量空间模型中每个查询词被看成是向量空间中的一个维度。因此可以将查询和文档看成是两个独立的向量。这两个向量的点积(scalar product)代表了文档和查询的相似度。这意味着文档中词的位置信息没有被使用，一个文档仅仅被看成一个词袋(a bag of words)。

一个简单的vsm模型的文档向量根据文档是否包含一个词将这个词对应的维度设成1或0。在不考虑查询词权重的情况下，查询向量中所有词对应的维度都被设成1。在这种简单模型下，两个向量的点积就是两个向量中公共词的个数。在这种简单模型下，两个包含同样数量公共词的文档，一篇每个词只出现一次，另一篇每个词出现多次，这两篇文档会获得相同的得分。然而哪篇文档的主题和查询词会更接近呢？tf/idf中的tf部分帮助我们解决这个问题：tf指的是词项的频率(term frequency)，解决方法就是使用词项的频率替代文档向量中简单的0或1。

最简单的计算tf的方法是将文档中出现某个词项的次数作为这个词项的tf。这种简单的方式会带来一个问题。假设一个查询是`fire fox`，两篇文档用tf表示分别是`D1{15,0}`和`D2{5,5}`,使用查询`Q{1,1}`查询的时候，D1得分会是15，D2得分是10。有可能D1是一篇很长的关于火灾的文章而D2是一篇关于如何安装火狐浏览器的入门教程。你认为哪篇更贴近于查询呢？为了让包含更多查询词的文档获得较高的得分，if/idf使用log(tf)作为向量维度的值。如果你还记得数学课学过的知识的话，应该知道log(0)无限趋于负无穷，而log(1)=0。为了获得想要的效果，我们可以在log运算前先给tf+1，也就是log(tf+1)。使用自然对数进行计算，现在两篇文档的向量变成了`D1{2.772588722239781,0}`和`D2{1.791759469228055,1.791759469228055}`，这样D1得分会是2.772588722239781而D2得分会是3.58351893845611。我们现在有了一个可以根据文档中所有词项频率评估文档相似度的模型。

tf-idf的另一个部分是IDF,逆文档频率(Inverse Document Frequency)。idf的定义是总的文档数/包含词项的文档数。**idf用来处理在一种语言中一部分词比其他词出现频率更高的现象。事实上很多频繁出现的词被作为噪音。如果在每篇文档中都含有同一个词，那么这个词对选择文档能有什么帮助呢？**你也许会说那么为什么不用停用词(stop words)呢？当然，Elasticsearch有停用词机制，然而使用停用词被证明有时可能会损失精确度，而且使用停用词不只是语言相关的，还是领域相关的。idf代表了一个词出现在文档中所代表的重要性。和tf相似，一般使用idf的log值。log所使用的基数是多少并不重要，重要的是它使得只有指数增长才能带来idf的实际增长。

为了避免除数为0，因此一般要在分数的分母处+1。在Lucene的TFIDFSimilarity中为了避免idf是负数一般还会在log外+1。

```
public float idf(long docFreq, long docCount) {
  return (float)(Math.log((docCount+1)/(double)(docFreq+1)) + 1.0);
}
```

```
/**
 * Create a PriorityQueue from a word-&gt;tf map.
 *
 * @param words a map of words keyed on the word(String) with Int objects as the values.
 * @param fieldNames an array of field names to override defaults.
 */
private PriorityQueue<ScoreTerm> createQueue(Map<String, Int> words, String... fieldNames) throws IOException {
 // have collected all words in doc and their freqs
 int numDocs = ir.numDocs();
  final int limit = Math.min(maxQueryTerms, words.size());
  FreqQ queue = new FreqQ(limit); // will order words by score

  for (String word : words.keySet()) { // for every word
  int tf = words.get(word).x; // term freq in the source doc
  if (minTermFreq > 0 && tf < minTermFreq) {
  continue; // filter out words that don't occur enough times in the source
  }

  // go through all the fields and find the largest document frequency
  String topField = fieldNames[0];
  int docFreq = 0;
  for (String fieldName : fieldNames) {
  int freq = ir.docFreq(new Term(fieldName, word));
  topField = (freq > docFreq) ? fieldName : topField;
  docFreq = (freq > docFreq) ? freq : docFreq;
  }

  if (minDocFreq > 0 && docFreq < minDocFreq) {
  continue; // filter out words that don't occur in enough docs
  }

  if (docFreq > maxDocFreq) {
  continue; // filter out words that occur in too many docs
  }

```

```
  if (docFreq == 0) {
  continue; // index update problem?
  }

  float idf = similarity.idf(docFreq, numDocs);
  float score = tf * idf;

  if (queue.size() < limit) {
  // there is still space in the queue
  queue.add(new ScoreTerm(word, topField, score, idf, docFreq, tf));
  } else {
  ScoreTerm term = queue.top();
  if (term.score < score) { // update the smallest in the queue in place and update the queue.
  term.update(word, topField, score, idf, docFreq, tf);
  queue.updateTop();
  }
  }
  }
  return queue;
 }
```