控制相　度

　理　　化数据（比如：　　、数字、字符串、枚　）的数据　，只需　　文　（或　系数据　里的行）是否与　　匹配。

布　的是/非匹配是全文搜索的基　，但不止如此，我　　要知道　个文　与　　的相　度，在全文搜索引中不　需要　到匹配的文　，　需根据它　相　度的高低　行排序。

全文相　的公式或　相似算法（*similarity　algorithms*）　会将多个因素合并起来，　　个文　生成一个相度　分 `_score` 。本章中，我　会　　各　可　部分，然后　　如何来控制它　。

当然，相　度不只与全文　　有　，也需要将　　化的数据考　其中。可能我　正在　一个度假屋，需要一些的　　特征（空　、海景、免　　WiFi　　），匹配的特征越多相　度越高。可能我希望有一些其他的考　因素，如回　率、　格、受　迎度或距　，当然也同　考　全文　　的相　度。

所有的　些都可以通　Elasticsearch　大的　分基　来　　。

本章会先从理　上介　Lucene 是如何　算相　度的，然后通　　例子　明如何控制相　度的　算　程。

相　度　分背后的理

Lucene（或　Elasticsearch）使用　布　模型（*Boolean　model*）　　匹配文　，并用一个名　　用　分函数（*practical scoring function*）的公式来　算相　度。　个公式借　了　　/逆向文　　率（*term frequency/inverse document frequency*）和 向量空　模型（*vector space model*），同　也加入了一些　代的新特性，如　　因子（coordination　　factor），字段　度　一化（field　　length normalization），以及　或　　句　重提升。

NOTE	不要　　！　些概念并没有像它　字面看起来那　　，尽管本小　提到了算法、公式和数学模型，但内容　是　人容易理解的，与理解算法本身相比，了解　些因素如何影　　果更　重要。

布　模型

布　模型（*Boolean　Model*）　只是在　　中使用 `AND` 、 `OR` 和 `NOT` （与、或和非）　　的条件来　匹配的文　，以下　　：

```
full AND text AND search AND (elasticsearch OR lucene)
```

会将所有包括 `full` 、 `text` 和 `search` ，以及 `elasticsearch` 或 `lucene` 的文　作　　果集。

　个　程　　且快速，它将所有可能不匹配的文　排除在外。

/逆向文　　率（TF/IDF）

当匹配到一　文　后，需要根据相　度排序　些文　，不是所有的文　都包含所有　，有些　比其他的　更重要。一个文　的相　度　分部分取决于　个　　在文　中的　重　。

　的　重由三个因素决定，在　什　是相　　中已　有所介　，有　趣可以了解下面的公式，但并不要求

住。

在文 中出 的 度是多少？ 度越高， 重 越高 。 5 次提到同一 的字段比只提到 1 次的更相 。 的 算方式如下：

```
tf(t in d) = √frequency ①
```

① t 在文 d 的 （ tf ）是 在文 中出 次数的平方根。

如果不在意 在某个字段中出 的 次，而只在意是否出 ， 可以在字段映射中禁用 ：

```
PUT /my_index
{
  "mappings": {
    "doc": {
      "properties": {
        "text": {
          "type":          "string",
          "index_options": "docs" ①
        }
      }
    }
  }
}
```

① 将参数 index_options 置 docs 可以禁用 及 位置， 个映射的字段不会 算 的出 次数， 于短 或近似 也不可用。要求精 的 not_analyzed 字符串字段会 使用 置。

逆向文 率

在集合所有文 里出 的 率是多少？ 次越高， 重 越低 。常用 如 and 或 the 相 度 献很少，因 它 在多数文 中都会出 ，一些不常 如 elastic 或 hippopotamus 可以 助我 快速 小 到感 趣的文 。逆向文 率的 算公式如下：

```
idf(t) = 1 + log ( numDocs / (docFreq + 1)) ①
```

① t 的逆向文 率（ idf ）是：索引中文 数量除以所有包含 的文 数，然后求其 数。

字段 度 一

字段的 度是多少？字段越短，字段的 重 越高 。如果 出 在 似 title 的字段，要比它出 在内容 body 的字段中的相 度更高。字段 度的 一 公式如下：

```
norm(d) = 1 / √numTerms ①
```

① 字段 度 一 （ norm ）是字段中 数平方根的倒数。

字段 度的 一 全文搜索非常重要， 多其他字段不需要有 一 。无 文 是否包括 个字段，索引中 个文 的 个 string 字段都大 占用 1 个 byte 的空 。 于 not_analyzed 字符串字段的 一 是禁用的，而 于 analyzed 字段也可以通 修改字段映射禁用 一 ：

```
PUT /my_index
{
  "mappings": {
    "doc": {
      "properties": {
        "text": {
          "type": "string",
          "norms": { "enabled": false } ①
        }
      }
    }
  }
}
```

① 个字段不会将字段 度 一 考 在内， 字段和短字段会以相同 度 算 分。

于有些 用 景如日志， 一 不是很有用，要 心的只是字段是否包含特殊的 或者特定的 器唯一 符。字段的 度 果没有影 ，禁用 一 可以 省大量内存空 。

合使用

以下三个因素——（term frequency）、逆向文 率（inverse document frequency）和字段度 一 （field-length norm）——是在索引 算并存 的。最后将它 合在一起 算 个在特定文 中的 重 。

TIP 前面公式中提到的 文 上是指文 里的某个字段，个字段都有它自己的倒排索引，因此字段的 TF/IDF 就是文 的 TF/IDF 。

当用 explain 看一个 的 term （参 explain ），可以 与 算相 度分的因子就是前面章 介 的 些：

```
PUT /my_index/doc/1
{ "text" : "quick brown fox" }

GET /my_index/doc/_search?explain
{
  "query": {
    "term": {
      "text": "fox"
    }
  }
}
```

以上 求（ 化）的 explanation 解 如下：

```
weight(text:fox in 0) [PerFieldSimilarity]:  0.15342641 ①
result of:
    fieldWeight in 0                         0.15342641
    product of:
        tf(freq=1.0), with freq of 1:        1.0 ②
        idf(docFreq=1, maxDocs=1):           0.30685282 ③
        fieldNorm(doc=0):                    0.5 ④
```

① fox 在文 的内部 Lucene doc ID 0，字段是 text 里的最 分。

② fox 在 文 text 字段中只出 了一次。

③ fox 在所有文 text 字段索引的逆向文 率。

④ 字段的字段 度 一 。

当然， 通常不止一个 ，所以需要一 合并多 重的方式——向量空 模型（vector space model）。

向量空 模型

向量空 模型（*vector space model*） 提供一 比 多 的方式， 个 分代表文 与 的匹配程度， 了做到 点， 个模型将文 和 都以 向量（*vectors*）的形式表示：

向量 上就是包含多个数的一 数 ，例如：

```
[1,2,5,22,3,8]
```

在向量空 模型里，向量空 模型里的 个数字都代表一个 的 重 ，与 /逆向文 率（term frequency/inverse document frequency） 算方式 似。

TIP 尽管 TF/IDF 是向量空 模型 算 重的 方式，但不是唯一方式。Elasticsearch 有其他模型如 Okapi-BM25 。TF/IDF 是 的因 它是个 的 又高效的算法，可以提供高 量的搜索 果。

想如果 “happy hippopotamus”，常 happy 的 重 低，不常 hippopotamus 重高，假 happy 的 重是 2， hippopotamus 的 重是 5，可以将 个二 向量—— [2,5] ——在坐 系下作条直 ， 的起点是 (0,0) 点是 (2,5)，如 表示 “happy hippopotamus” 的二 向量 。

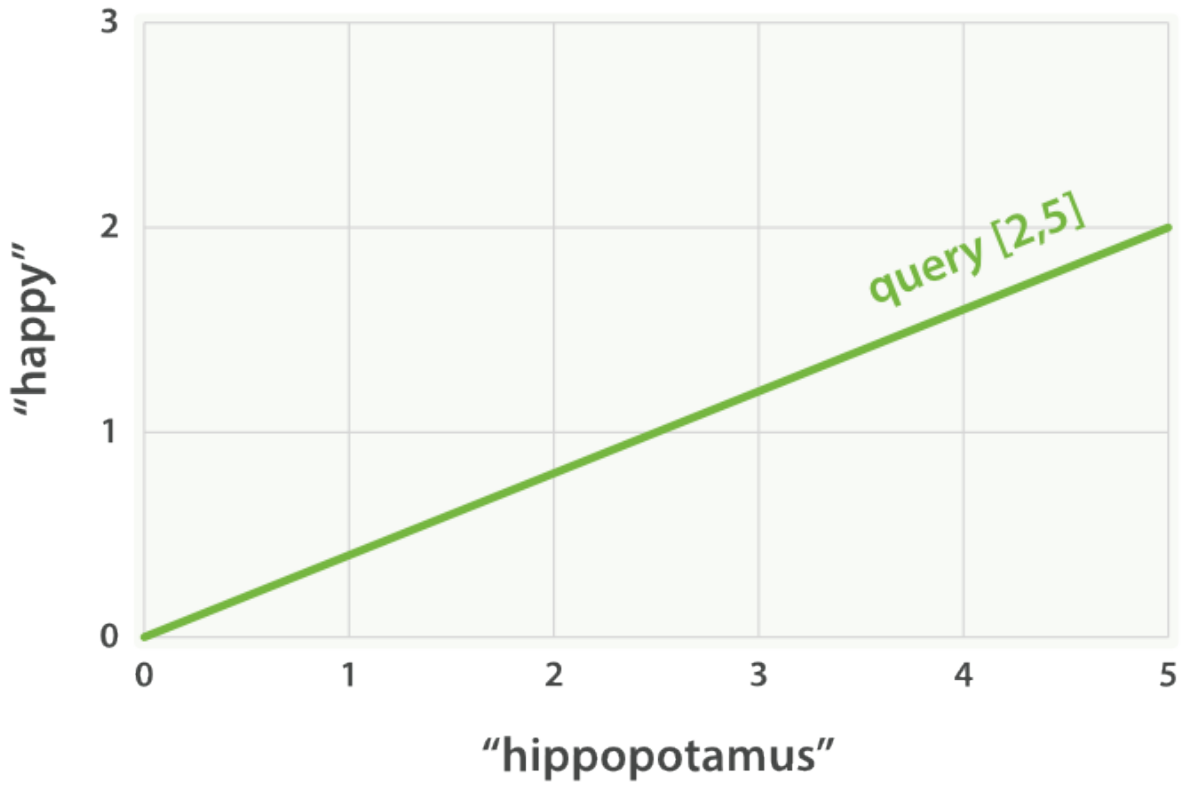

Figure 1. 表示 *"happy hippopotamus"* 的二 向量

在， 想我 有三个文 ：

1. I am *happy* in summer 。
2. After Christmas I'm a *hippopotamus* 。
3. The *happy hippopotamus* helped Harry 。

可以 个文 都 建包括 个 —— `happy` 和 `hippopotamus` —— 重的向量，然后将 些向量置入同一个坐 系中，如 "happy hippopotamus" 及文 向量 ：

- 文 1： `(happy,__)` —— `[2,0]`
- 文 2： `( _ ,hippopotamus)` —— `[0,5]`
- 文 3： `(happy,hippopotamus)` —— `[2,5]`

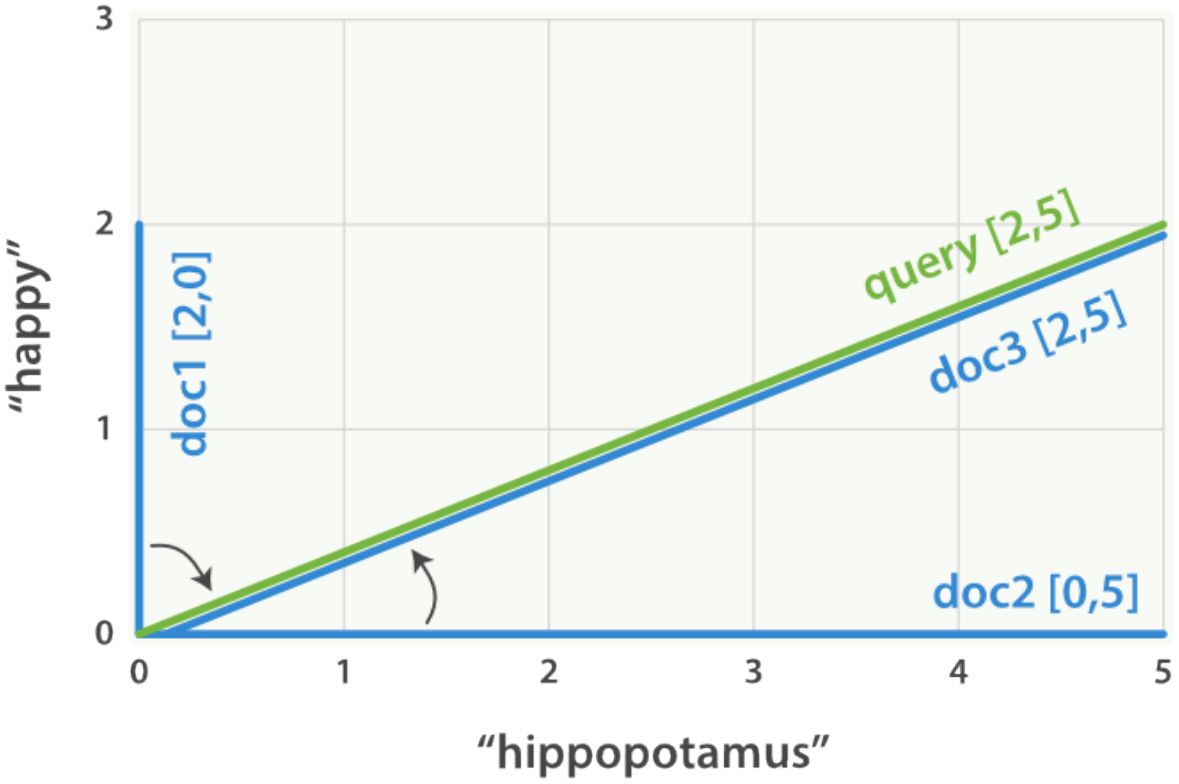

Figure 2. "happy hippopotamus" 及文 向量

向量之 是可以比 的，只要 量 向量和文 向量之 的角度就可以得到 个文 的相 度，文 1 与 之 的角度最大，所以相 度低；文 2 与 的角度 小，所以更相 ；文 3 与 的角度正好吻合，完全匹配。

TIP

在 中，只有二 向量（ 个 的 ）可以在平面上表示，幸 的是， 性代数 —— 作 数学中 理向量的一个分支—— 我 提供了 算 个多 向量 角度工具， 意味着可以使用如上同 的方式来解 多个 的 。

于比 个向量的更多信息可以参考 余弦近似度（*cosine similarity*）。

在已 完 分 算的基本理 ，我 可以 了解 Lucene 是如何 分 算的。

Lucene 的 用 分函数

于多 ， Lucene 使用 布 模型（Boolean model） 、 TF/IDF 以及 向量空 模型（vector space model），然后将它 合到 个高效的包里以收集匹配文 并 行 分 算。

一个多

```
GET /my_index/doc/_search
{
  "query": {
    "match": {
      "text": "quick fox"
    }
  }
}
```

会在内部被重写 ：

```
GET /my_index/doc/_search
{
  "query": {
    "bool": {
      "should": [
        {"term": { "text": "quick" }},
        {"term": { "text": "fox"   }}
      ]
    }
  }
}
```

bool 了布 模型，在 个例子中，它会将包括 quick 和 fox 或 者兼有的文 作 果。

只要一个文 与 匹配，Lucene 就会 算 分，然后合并 个匹配 的 分 果。 里使用的 分 算公式叫做 用 分函数（*practical scoring function*） 。看似很高大上，但是 被 到——多数的 件都已 介 ，下一 会 它引入的一些新元素。

```
score(q,d)  =  ①
            queryNorm(q)  ②
          · coord(q,d)    ③
          · ∑ (           ④
                tf(t in d)   ⑤
              · idf(t)²      ⑥
              · t.getBoost() ⑦
              · norm(t,d)    ⑧
            ) (t in q)    ④
```

① score(q,d) 是文 d 与 q 的相 度 分。

② queryNorm(q) 是 一化因子 （新）。

③ coord(q,d) 是 因子 （新）。

④ q 中 个 t 于文 d 的 重和。

⑤ tf(t in d) 是 t 在文 d 中的 。

⑥ idf(t) 是 t 的逆向文 率。

⑦ t.getBoost() 是　　中使用的 *boost*（新）。

⑧ norm(t,d) 是 字段　度　一　，与 索引　字段　boost （如果存在）的和（新）。

上　已介　　score 、 tf 和 idf 。　在来介　queryNorm 、 coord 、 t.getBoost 和 norm 。

我　会在本章后面　　探　　的　重提升　　的　　，但是首先需要了解　　一化、　　和索引字段　面的　重提升等概念。

一因子

一因子 （ queryNorm ）　　将　　一化，　　就能将　个不同的　　果相比　。

> **TIP**
> 尽管　　一　的目的是　了使　　果之　能　相互比　，但是它并不十分有效，因　相度　分　　_score　　的目的是　了将当前　　的　果　行排序，比　不同　　果的相　度　分没有太大意　。

个因子是在　　程的最前面　算的，具体的　算依　于具体　　，一个典型的　　如下：

```
queryNorm = 1 / √sumOfSquaredWeights ①
```

① sumOfSquaredWeights 是　　里　个　的 IDF 的平方和。

> **TIP**
> 相同　　一化因子会被　用到　个文　，不能被更改，　而言之，可以被忽略。

因子　　（　　coord　　）可以　那些　　包含度高的文　提供　励，文　里出　的越多，它越有机会成　好的匹配　果。

想　　quick　brown　fox　，　个　的　重都是　1.5　。如果没有　　因子，最　　分会是文　里所有　重的　和。例如：

- 文　里有 fox →　分： 1.5
- 文　里有 quick fox →　分： 3.0
- 文　里有 quick brown fox →　分： 4.5

因子将　分与文　里匹配　的数量相乘，然后除以　　里所有　的数量，如果使用　　因子，　分会成：

- 文　里有 fox →　分： 1.5 * 1 / 3 = 0.5
- 文　里有 quick fox →　分： 3.0 * 2 / 3 = 2.0
- 文　里有 quick brown fox →　分： 4.5 * 3 / 3 = 4.5

因子能使包含所有三个　的文　比只包含　个　的文　　分要高出很多。

回想将　　quick brown fox 重写成 bool　　的形式：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "term": { "text": "quick" }},
        { "term": { "text": "brown" }},
        { "term": { "text": "fox"   }}
      ]
    }
  }
}
```

bool 会 所有 should 句使用 功能，不 也可以将其禁用。 什 要做？通常的回答是——无 。 通常是件好事，当使用 bool 将多个高 如 match 包 的 候， 功能 是有意 的，匹配的 句越多， 求与返回文 的重 度就越高。

但在某些高 用中，将 功能 可能更好。 想正在 同 jump 、 leap 和 hop ，并不 心会出 多少个同 ，因 它 都表示相同的意思， 上，只有其中一个同 会出 ， 是不使用 因子的一个好例子：

```
GET /_search
{
  "query": {
    "bool": {
      "disable_coord": true,
      "should": [
        { "term": { "text": "jump" }},
        { "term": { "text": "hop"  }},
        { "term": { "text": "leap" }}
      ]
    }
  }
}
```

当使用同 的 候（参照： 同 ），Lucene 内部是 的：重写的 会禁用同 的 功能。大多数禁用操作的 用 景是自 理的，无 此担心。

索引 字段 重提升

我 会 的 重提升， 字段 重提升 就是 某个字段比其他字段更重要。当然在索引 也能做到如此。 上， 重的提升会被 用到字段的 个 ，而不是字段本身。

将提升 存 在索引中无 更多空 ， 个字段 索引 的提升 与字段 度 一 （参 字段 度 一 ）一起作 个字 存于索引， norm(t,d) 是前面公式的返回 。

WARNING

我 不建 在建立索引 字段提升 重，有以下原因：

- 将提升 与字段 度 一 合在 个字 中存 会 失字段 度 一 的精度，会 致 Elasticsearch 不知如何区分包含三个 的字段和包含五个 的字段。
- 要想改 索引 的提升 ，就必 重新 所有文 建立索引，与此不同的是，的提升 可以随着 次 的不同而更改。
- 如果一个索引 重提升的字段有多个 ，提升 会按照 个 来自乘， 会 致字段的 重急 上升。

予 重 是更 、清楚、 活的 。

了解了 一化、 同和索引 重提升 些方式后，可以 一 了解相 度 算最有用的工具：的 重提升。

重提升

在 句 先 （Prioritizing Clauses） 中，我 解 如何在搜索 使用 boost 参数 一个 句比其他 句更重要。例如：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        {
          "match": {
            "title": {
              "query": "quick brown fox",
              "boost": 2 ①
            }
          }
        },
        {
          "match": { ②
            "content": "quick brown fox"
          }
        }
      ]
    }
  }
}
```

① title 句的重要性是 content 的 2 倍，因 它的 重提升 2 。

② 没有 置 boost 的 句的 1 。

的 重提升 是可以用来影 相 度的主要工具，任意 型的 都能接受 boost 参数。将 boost 置 2 ，并不代表最 的 分 _score 是原 的 倍； 的 重 会 一化和一些其他内部 化

程。尽管如此，它 想要表明一个提升 2 的句子的重要性是提升 1 句的 倍。

在 用中，无法通 的公式得出某个特定 句的 正 '' 重提升 ，只能通 不断 得。需要 住的是 boost 只是影 相 度 分的其中一个因子；它 需要与其他因子相互 争。在前例中， title 字段相 content 字段可能已 有一个 省的" 重提升 ， 因 在 字段度 一 中， 往往比相 内容要短，所以不要想当然的去盲目提升一些字段的 重。 重， 果，如此反 。

提升索引 重

当在多个索引中搜索 ，可以使用参数 indices_boost 来提升整个索引的 重，在下面例子中，当要 最近索引的文 分配更高 重 ，可以 做：

```
GET /docs_2014_*/_search ①
{
  "indices_boost": { ②
    "docs_2014_10": 3,
    "docs_2014_09": 2
  },
  "query": {
    "match": {
      "text": "quick brown fox"
    }
  }
}
```

① 个多索引 涵 了所有以字符串 docs_2014_ 始的索引。

② 其中，索引 docs_2014_10 中的所有文件的 重是 3 ，索引 docs_2014_09 中是 2 ，其他所有匹配的索引 重 1 。

t.getBoost()

些提升 在 Lucene 的 用 分函数 中可以通 t.getBoost() 得。 重提升不会被 用于它在 表 式中出 的 ，而是会被合并下 至 个 中。 t.getBoost() 始 返回当前 的 重或当前分析 上 的 重。

TIP

上，要想解 explain 的 出是相当 的，在 explanation 里面完全看不到 boost ，也完全无法 上面提到的 t.getBoost() 方法， 重 融合在 queryNorm 中并 用到 个 。尽管 ， queryNorm 于 个 都是相同的， 是会 一个 重提升 的 的 queryNorm 要高于一个没有提升 的。

使用 修改相 度

Elasticsearch 的 表 式相当 活，可以通 整 中 句的所 次，从而或多或少改 其重要性，比如， 想下面 个 ：

```
quick OR brown OR red OR fox
```

可以将所有 都放在 bool 的同一 中：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "term": { "text": "quick" }},
        { "term": { "text": "brown" }},
        { "term": { "text": "red"   }},
        { "term": { "text": "fox"   }}
      ]
    }
  }
}
```

个 可能最 包含 quick 、 red 和 brown 的文 分与包含 quick 、 red 、 fox 文 的 分相同，里 *Red* 和 *brown* 是同 ，可能只需要保留其中一个，而我 真正要表 的意思是想做以下 ：

```
quick OR (brown OR red) OR fox
```

根据 准的布 ， 与原始的 是完全一 的，但是我 已 在 合 （Combining Queries）中看到， bool 不 心文 匹配的 程度，只 心是否能匹配。

上述 有个更好的方式：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "term": { "text": "quick" }},
        { "term": { "text": "fox"   }},
        {
          "bool": {
            "should": [
              { "term": { "text": "brown" }},
              { "term": { "text": "red"   }}
            ]
          }
        }
      ]
    }
  }
}
```

在， red 和 brown 于相互 争的 次， quick 、 fox 以及 red OR brown 是 于 且相互 争的 。

我 已 如何使用 match 、multi_match 、term 、bool 和 dis_max 修改相 度 分。本章后面的内容会介 外三个与相 度 分有 的 ： boosting 、 constant_score 和 function_score 。

Not Quite Not

在互 上搜索 “Apple”，返回的 果很可能是一个公司、水果和各 食 。我 可以在 bool 中用 must_not 句来排除像 pie 、 tart 、 crumble 和 tree 的 ，从而将 果的 小至只返回与 “Apple”（ 果）公司相 的 果：

```
GET /_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "text": "apple"
        }
      },
      "must_not": {
        "match": {
          "text": "pie tart fruit crumble tree"
        }
      }
    }
  }
}
```

但 又敢保 在排除 tree 或 crumble 后，不会 失一个与 果公司特 相 的文 ？有 ，must_not 条件会 于 格。

重提升

{ref}/query-dsl-boosting-query.html[boosting] 恰恰能解决 个 。它 然允 我 将 于水果或甜点的 果包括到 果中，但是使它 降 ——即降低它 原来可能 有的排名：

```
GET /_search
{
  "query": {
    "boosting": {
      "positive": {
        "match": {
          "text": "apple"
        }
      },
      "negative": {
        "match": {
          "text": "pie tart fruit crumble tree"
        }
      },
      "negative_boost": 0.5
    }
  }
}
```

它接受 positive 和 negative 。只有那些匹配 positive 的文 列出来， 于那些同 匹配 negative 的文 将通 文 的原始 _score 与 negative_boost 相乘的方式降 后的 果。

了 到效果， negative_boost 的 必 小于 1.0 。在 个示例中，所有包含 向 的文 分 _score 都会 半。

忽略 TF/IDF

有 候我 根本不 心 TF/IDF ，只想知道一个 是否在某个字段中出 。可能搜索一个度假屋并希望它能尽可能有以下 施：

- WiFi
- Garden（花 ）
- Pool（游泳池）

个度假屋的文 如下：

```
{ "description": "A delightful four-bedroomed house with ... " }
```

可以用 的 match 行匹配：

```
GET /_search
{
  "query": {
    "match": {
      "description": "wifi garden pool"
    }
  }
}
```

但 并不是真正的 全文搜索 ，此 情况下，TF/IDF 并无用 。我 既不 心 wifi 是否 一个普通 ，也不 心它在文 中出 是否 繁， 心的只是它是否曾出 。 上，我 希望根据房屋不同 施的数量 其排名—— 施越多越好。如果 施出 ， 1 分，不出 0 分。

constant_score

在 {ref}/query-dsl-constant-score-query.html[constant_score] 中，它可以包含 或 ， 任意一个匹配的文 指定 分 1 ，忽略 TF/IDF 信息：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "constant_score": {
          "query": { "match": { "description": "wifi" }}
        }},
        { "constant_score": {
          "query": { "match": { "description": "garden" }}
        }},
        { "constant_score": {
          "query": { "match": { "description": "pool" }}
        }}
      ]
    }
  }
}
```

或 不是所有的 施都同等重要—— 某些用 来 有些 施更有 。如果最重要的 施是游泳池，那我 可以 更重要的 施 加 重：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "constant_score": {
          "query": { "match": { "description": "wifi" }}
        }},
        { "constant_score": {
          "query": { "match": { "description": "garden" }}
        }},
        { "constant_score": {
          "boost":   2 ①
          "query": { "match": { "description": "pool" }}
        }}
      ]
    }
  }
}
```

① pool 句的 重提升 2，而其他的 句 1。

NOTE	最 的 分并不是所有匹配 句的 求和， 因子（coordination factor） 和 一化因子（query normalization factor） 然会被考 在内。

我 可以 features 字段加上 not_analyzed 型来提升度假屋文 的匹配能力：

```
{ "features": [ "wifi", "pool", "garden" ] }
```

情况下，一个 not_analyzed 字段会禁用 字段 度 — （field-length norms） 的功能，并将 index_options docs ，禁用 ，但 是存在 ： 个 的 倒排文 率 然会被考 。

可以采用与之前相同的方法 constant_score 来解决 个 ：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "constant_score": {
          "query": { "match": { "features": "wifi" }}
        }},
        { "constant_score": {
          "query": { "match": { "features": "garden" }}
        }},
        { "constant_score": {
          "boost":   2
          "query": { "match": { "features": "pool" }}
        }}
      ]
    }
  }
}
```

上， 个 施都 看成一个 器， 于度假屋来 要 具有某个 施要 没有—— 器因 其性天然合 。而且，如果使用 器，我 可以利用 存。

里的 是： 器无法 算 分。 就需要 求一 方式将 器和 的差 抹平。 function_score 不 正好可以扮演 个角色，而且有更 大的功能。

function_score

{ref}/query-dsl-function-score-query.html[function_score] 是用来控制 分 程的武器，它允 个与主 匹配的文 用一个函数，以 到改 甚至完全替 原始 分 _score 的目的。

上，也能用 器 果的 子集 用不同的函数， 一箭双 ：既能高效 分，又能利用 器存。

Elasticsearch 定 了一些函数：

weight
个文 用一个 而不被 化的 重提升 ：当 weight 2 ，最 果 2 * _score 。

field_value_factor

使用 个 来修改 _score，如将 popularity 或 votes （受 迎或 ）作 考 因素。

random_score

个用 都使用一个不同的随机 分 果排序，但 某一具体用 来 ，看到的 序始 是一致的。

衰 函数 —— linear 、 exp 、 gauss

将浮 合到 分 _score 中，例如 合 publish_date 得最近 布的文 ， 合 geo_location 得更接近某个具体 度（lat/lon）地点的文 ， 合 price 得更接近某个特定 格的文 。

script_score

如果需求超出以上 ，用自定 脚本可以完全控制 分 算， 所需 。

如果没有 function_score ，就不能将全文 与最新 生 因子 合在一起 分，而不得不根据 分 _score 或 date 行排序； 会相互影 抵消 排序各自的效果。 个 可以使 个效果融合：可以 然根据全文相 度 行排序，但也会同 考 最新 布文 、流行文 、或接近用 希望 格的 品。正如所 想的， 要考 所有 些因素会非常 ， 我 先从 的例子 始，然后 着梯子慢慢向上爬， 加 度。

按受 迎度提升 重

想有个 站供用 布博客并且可以 他 自己喜 的博客点 ，我 希望将更受 迎的博客放在搜索 果列表中相 上的位置，同 全文搜索的 分 然作 相 度的主要排序依据，可以 的通 存 个博客的点 数来 它：

```
PUT /blogposts/post/1
{
  "title":   "About popularity",
  "content": "In this post we will talk about...",
  "votes":   6
}
```

在搜索 ，可以将 function_score 与 field_value_factor 合使用，即将点 数与全文相 度 分 合：

```
GET /blogposts/post/_search
{
  "query": {
    "function_score": { ①
      "query": { ②
        "multi_match": {
          "query":    "popularity",
          "fields": [ "title", "content" ]
        }
      },
      "field_value_factor": { ③
        "field": "votes" ④
      }
    }
  }
}
```

① function_score 将主 和函数包括在内。

② 主 先 行。

③ field_value_factor 函数会被 用到 个与主 query 匹配的文 。

④ 个文 的 votes 字段都 必 有 供 function_score 算。如果 没有 文 的 votes 字段有 ，那 就 必 使用 {ref}/query-dsl-function-score-query.html#function-field-value-factor[missing 属性] 提供的 来 行 分 算。

在前面示例中， 个文 的最 分 _score 都做了如下修改：

```
new_score = old_score * number_of_votes
```

然而 并不会 来出人意料的好 果，全文 分 _score 通常 于 0 到 10 之 ，如下 受 迎度的 性 系基于 _score 的原始 2.0 中，有 10 个 的博客会掩 掉全文 分，而 0 个 的博客的 分会被置 0 。

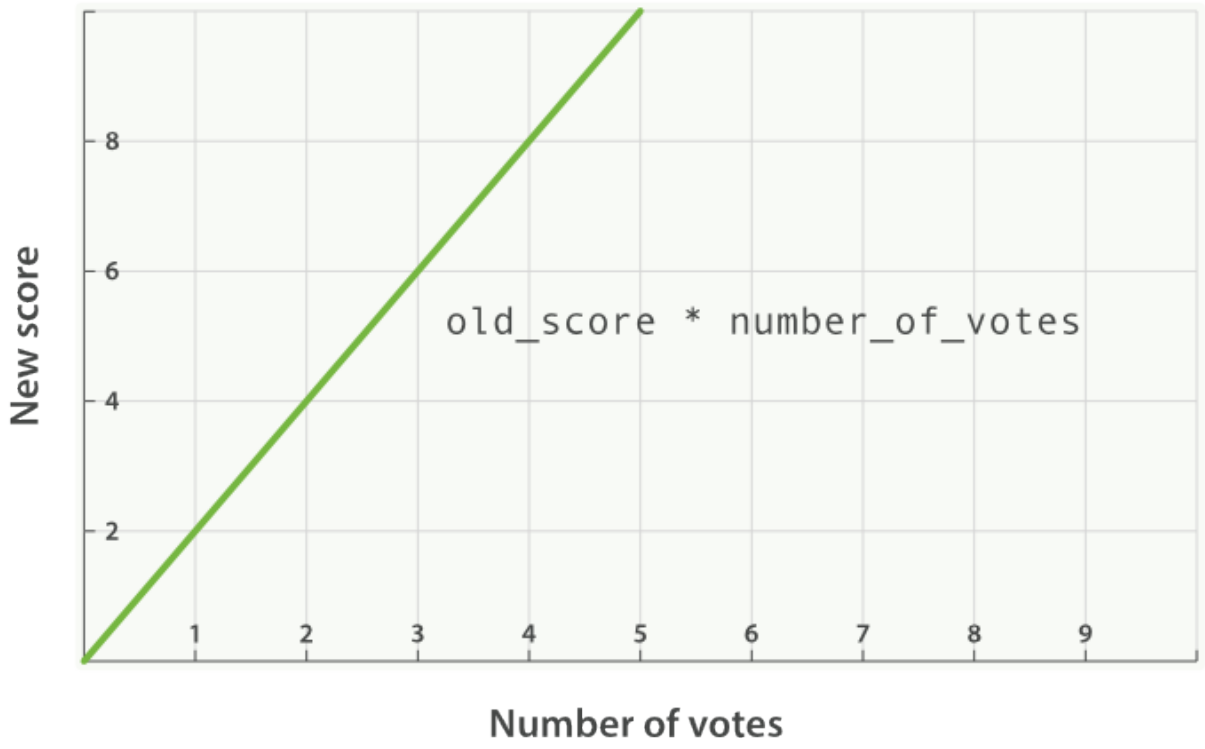

Figure 3. 受　迎度的　性　系基于 `_score` 的原始　`2.0`

modifier

一　融入受　迎度更好方式是用　`modifier`　平滑　`votes`　的　。　句　，我　希望最　始的一些更重要，但是其重要性会随着数字的　加而降低。 0 个　与 1 个　的区　比 10 个　与 11 个　的区大很多。

于上述情况，典型的 `modifier`　用是使用 `log1p` 参数　，公式如下：

```
new_score = old_score * log(1 + number_of_votes)
```

`log`　数函数使 `votes`　字段的　分曲　更平滑，如　受　迎度的　数　系基于 `_score` 的原始　`2.0` ：

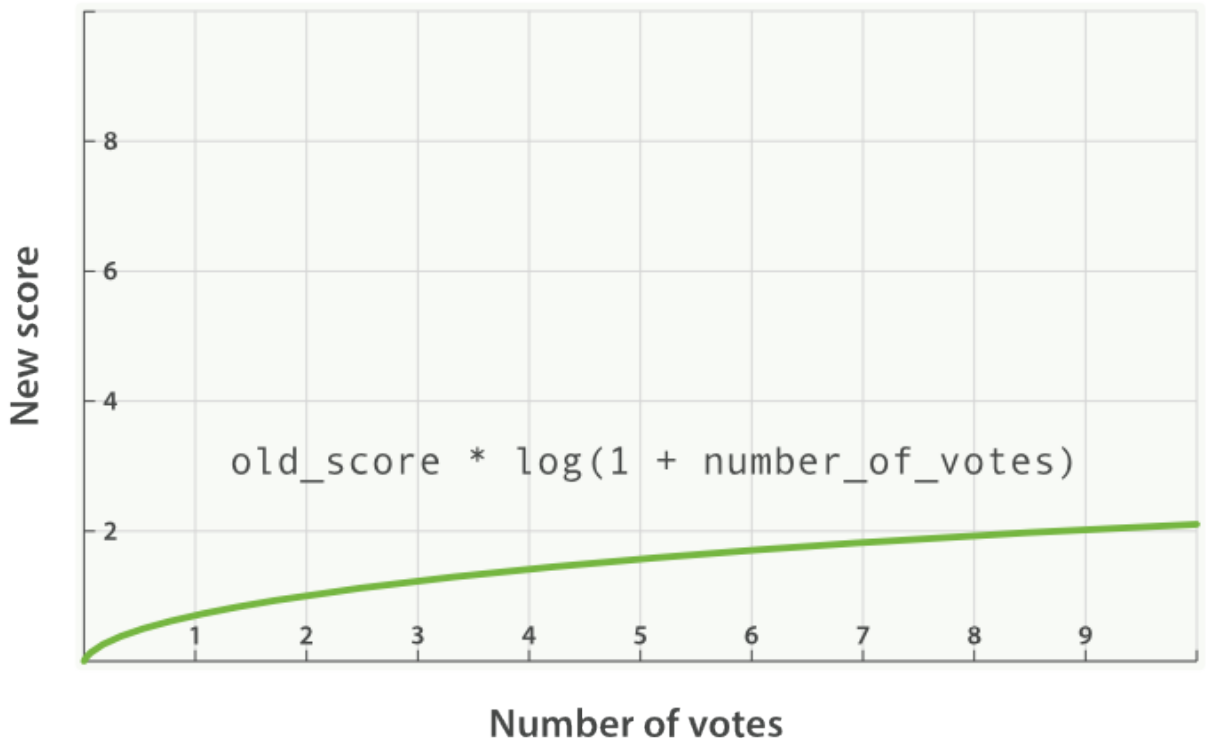

Figure 4. 受　迎度的　数　系基于 `_score` 的原始　`2.0`

`modifier` 参数的　求如下：

```
GET /blogposts/post/_search
{
  "query": {
    "function_score": {
      "query": {
        "multi_match": {
          "query":    "popularity",
          "fields": [ "title", "content" ]
        }
      },
      "field_value_factor": {
        "field":    "votes",
        "modifier": "log1p" ①
      }
    }
  }
}
```

① `modifier`　`log1p`。

修　modifier 的　可以　：`none`（　状　）、`log`、`log1p`、`log2p`、`ln`、`ln1p`、`ln2p`、`square`、　`sqrt`　以及　`reciprocal`　。想要了解更多信息　参照：　{ref}/query-dsl-function-score-query.html#function-field-value-factor[`field_value_factor` 文　].

factor

可以通　将 votes 字段与 factor 的　来　　受　迎程度效果的高低：

```
GET /blogposts/post/_search
{
  "query": {
    "function_score": {
      "query": {
        "multi_match": {
          "query":    "popularity",
          "fields": [ "title", "content" ]
        }
      },
      "field_value_factor": {
        "field":    "votes",
        "modifier": "log1p",
        "factor":   2 ①
      }
    }
  }
}
```

① 双倍效果。

添加了 factor 会使公式　成　　：

```
new_score = old_score * log(1 + factor * number_of_votes)
```

factor　大于 1 会提升效果， factor　小于 1 会降低效果，如　受　迎度的　数　系基于多个不同因子 。

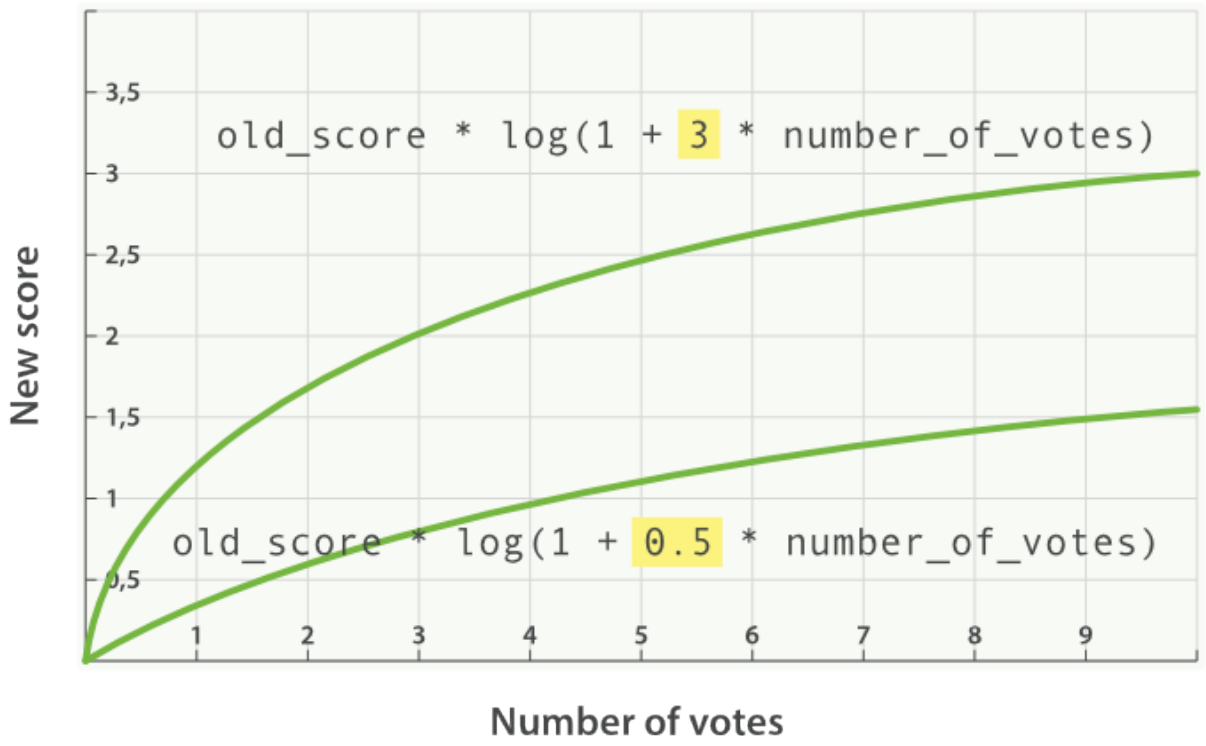

Figure 5. 受 迎度的 数 系基于多个不同因子

boost_mode

或 将全文 分与 `field_value_factor` 函数 乘 的效果 然可能太大，我 可以通 参数 `boost_mode` 来控制函数与 分 `_score` 合并后的 果，参数接受的 ：

`multiply`
分 `_score` 与函数 的 （ ）

`sum`
分 `_score` 与函数 的和

`min`
分 `_score` 与函数 的 小

`max`
分 `_score` 与函数 的 大

`replace`
函数 替代 分 `_score`

与使用乘 的方式相比，使用 分 `_score` 与函数 求和的方式可以弱化最 效果，特 是使用一个 小 `factor` 因子 ：

```
GET /blogposts/post/_search
{
  "query": {
    "function_score": {
      "query": {
        "multi_match": {
          "query":    "popularity",
          "fields": [ "title", "content" ]
        }
      },
      "field_value_factor": {
        "field":    "votes",
        "modifier": "log1p",
        "factor":   0.1
      },
      "boost_mode": "sum" ①
    }
  }
}
```

① 分 _score 与函数 的 。

之前 求的公式 在 成下面 （参 使用 sum 合受 迎程度 ）：

```
new_score = old_score + log(1 + 0.1 * number_of_votes)
```

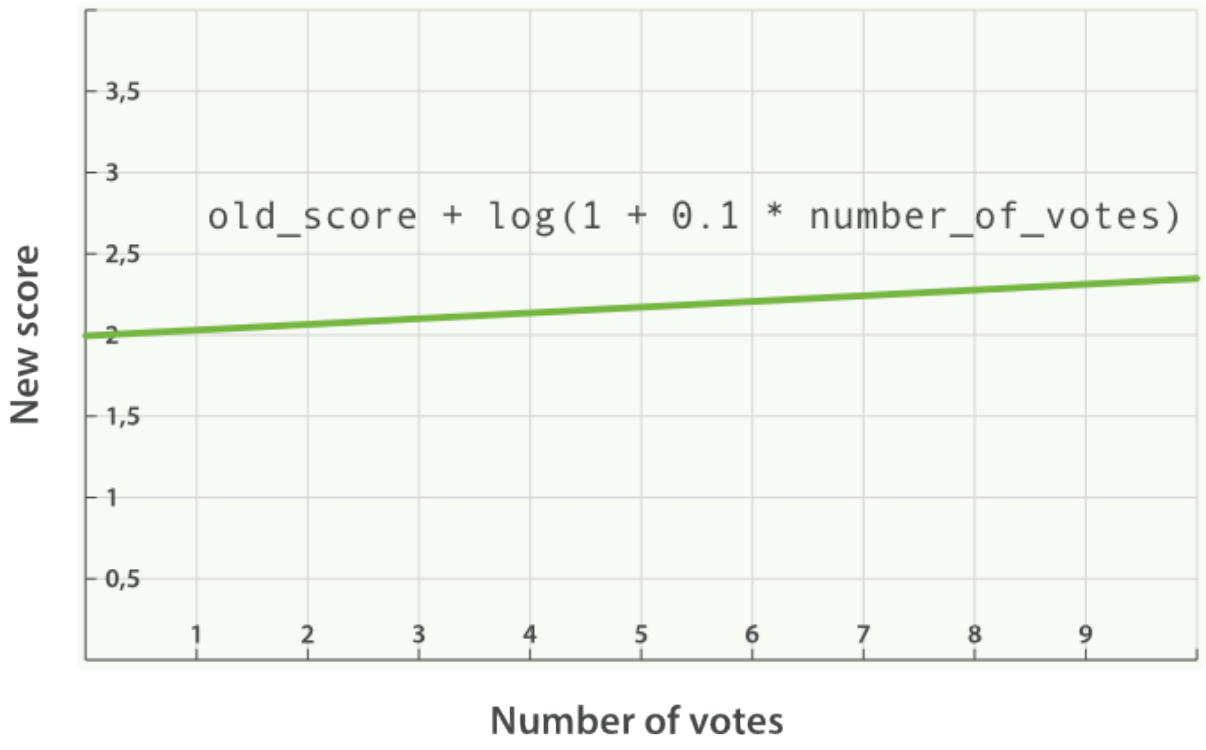

Figure 6. 使用 sum 合受 迎程度

max_boost

最后，可以使用 max_boost 参数限制一个函数的最大效果：

```
GET /blogposts/post/_search
{
  "query": {
    "function_score": {
      "query": {
        "multi_match": {
          "query":    "popularity",
          "fields": [ "title", "content" ]
        }
      },
      "field_value_factor": {
        "field":    "votes",
        "modifier": "log1p",
        "factor":   0.1
      },
      "boost_mode": "sum",
      "max_boost":  1.5 ①
    }
  }
}
```

① 无 field_value_factor 函数的 果如何，最 果都不会大于 1.5 。

NOTE max_boost 只 函数的 果 行限制，不会 最 分 _score 生直接影 。

集提升 重

回到 忽略 TF/IDF 里 理 的 ，我 希望根据 个度假屋的特性数量来 分，当 我 希望能用存的 器来影 分， 在 function_score 正好可以完成 件事情。

到目前 止，我 展 的都是 所有文 用 个函数的使用方式， 在会用 器将 果 分 多个子集（ 个特性一个 器），并 个子集使用不同的函数。

在下面例子中，我 会使用 weight 函数，它与 boost 参数 似可以用于任何 。有一点区 是 weight 没有被 Luence 一化成 以理解的浮点数，而是直接被 用。

的 需要做相 更以整合多个函数：

```
GET /_search
{
  "query": {
    "function_score": {
      "filter": { ①
        "term": { "city": "Barcelona" }
      },
      "functions": [ ②
        {
          "filter": { "term": { "features": "wifi" }}, ③
          "weight": 1
        },
        {
          "filter": { "term": { "features": "garden" }}, ③
          "weight": 1
        },
        {
          "filter": { "term": { "features": "pool" }}, ③
          "weight": 2 ④
        }
      ],
      "score_mode": "sum", ⑤
    }
  }
}
```

① function_score 有个 filter 器而不是 query 。

② functions 字存 着一个将被 用的函数列表。

③ 函数会被 用于和 filter 器（可 的）匹配的文 。

④ pool 比其他特性更重要，所以它有更高 weight 。

⑤ score_mode 指定各个函数的 行 合 算的方式。

个新特性需要注意的地方会在以下小 介 。

vs.

首先要注意的是 filter 器代替了 query ，在本例中，我 无 使用全文搜索，只想 到 city 字段中包含 Barcelona 的所有文 ， 用 比用 表 更清晰。 器返回的所有文 的 分 _score 的 1 。 function_score 接受 query 或 filter ，如果没有特 指定， 使用 match_all 。

函数 functions

functions 字保持着一个将要被使用的函数列表。可以 列表里的 个函数都指定一个 filter 器，在 情况下，函数只会被 用到那些与 器匹配的文 ，例子中，我 与 器匹配的文 指定 重 weight 1 （ 与 pool 匹配的文 指定 重 2 ）。

分模式 score_mode

个函数返回一个 果，所以需要一 将多个 果 到 个 的方式，然后才能将其与原始 分 _score 合并。 分模式 score_mode 参数正好扮演 的角色，它接受以下 ：

multiply
函数 果求 （ ）。

sum
函数 果求和。

avg
函数 果的平均 。

max
函数 果的最大 。

min
函数 果的最小 。

first
使用首个函数（可以有 器，也可能没有）的 果作 最 果

在本例中，我 将 个 器匹配 果的 重 weight 求和，并将其作 最 分 果，所以会使用 sum 分模式。

不与任何 器匹配的文 会保有其原始 分，_score 的 1 。

随机 分

可能会想知道 一致随机 分（*consistently random scoring*） 是什 ，又 什 会使用它。之前的例子是个很好的 用 景，前例中所有的 果都会返回 1 、 2 、 3 、 4 或 5 的最 分 _score ，可能只有少数房子的 分是 5 分，而有大量房子的 分是 2 或 3 。

作 站的所有者， 会希望 广告有更高的展 率。在当前 下，有相同 分 _score 的文 会 次都以相同次序出 ， 了提高展 率，在此引入一些随机性可能会是个好主意， 能保 有相同 分的文 都能有均等相似的展 机率。

我 想 个用 看到不同的随机次序，但也同 希望如果是同一用 翻 ， 果的相 次序能始 保持一致。 行 被称 一致随机（*consistently random*） 。

random_score 函数会 出一个 0 到 1 之 的数，当 子 seed 相同 ，生成的随机 果是一致的，例如，将用 的会 ID 作 seed ：

```
GET /_search
{
  "query": {
    "function_score": {
      "filter": {
        "term": { "city": "Barcelona" }
      },
      "functions": [
        {
          "filter": { "term": { "features": "wifi" }},
          "weight": 1
        },
        {
          "filter": { "term": { "features": "garden" }},
          "weight": 1
        },
        {
          "filter": { "term": { "features": "pool" }},
          "weight": 2
        },
        {
          "random_score": { ①
            "seed":  "the users session id" ②
          }
        }
      ],
      "score_mode": "sum"
    }
  }
}
```

① random_score 句没有任何 器 filter，所以会被 用到所有文 。

② 将用 的会 ID 作 子 seed， 用 的随机始 保持一致，相同的 子 seed 会 生相同的随机 果。

当然，如果 加了与 匹配的新文 ，无 是否使用一致随机，其 果 序都会 生 化。

越近越好

很多 量都可以影 用 于度假屋的 ，也 用 希望 市中心近点，但如果 格足 便宜，也有可能 一个更 的住 ，也有可能反 来是正 的： 意 最好的位置付更多的 。

如果我 添加 器排除所有市中心方 1 千米以外的度假屋，或排除所有 格超 £100 英的，我 可能会将用 意考 妥 的那些 排除在外。

function_score 会提供一 衰 函数（*decay functions*） ， 我 有能力在 个滑准，如地点和 格，之 衡。

有三 衰 函数—— linear 、 exp 和 gauss （ 性、指数和高斯函数），它 可以操作数 、

以及　　度地理坐　点　　的字段。所有三个函数都能接受以下参数：

`origin`
　中心点 或字段可能的最佳　，落在原点 `origin` 上的文　　分 `_score`　　分 `1.0` 。

`scale`
　衰　率，即一个文　从原点 `origin` 下落　，　分 `_score` 改　的速度。（例如，　£10 欧元或　100 米）。

`decay`
　从原点 `origin` 衰　到 `scale` 所得的　分 `_score`，　　`0.5` 。

`offset`
　以原点 `origin`　中心点，　其　置一个非零的偏移量 `offset` 覆　一个　　，而不只是　个原点。在 `-offset ⇐ origin ⇐ +offset` 内的所有　分 `_score` 都是 `1.0` 。

三个函数的唯一区　就是它　衰　曲　的形状，用　来　明会更　直　（参　衰　函数曲　）。

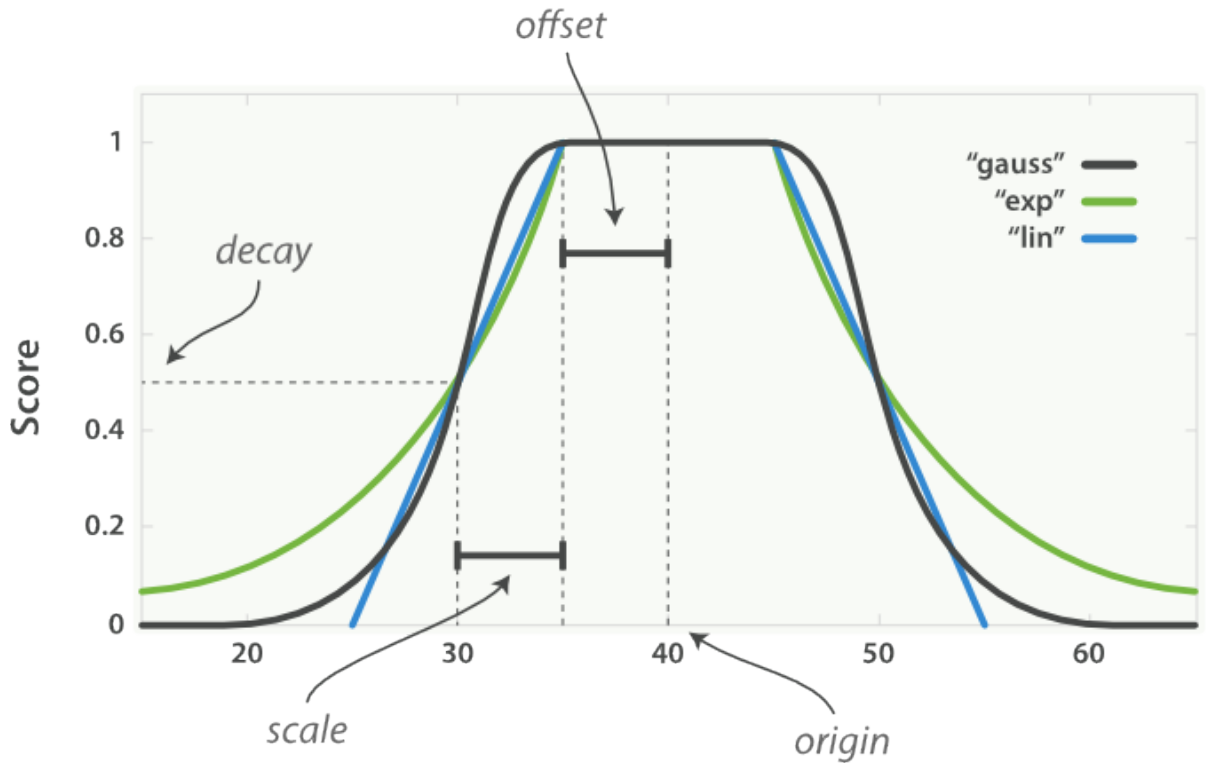

Figure 7. 衰　函数曲

衰　函数曲　中所有曲　的原点 `origin`（即中心点）的　都是 `40`，`offset` 是 `5`，也就是在　`40 - 5 ⇐ value ⇐ 40 + 5` 内的所有　都会被当作原点 `origin`　理——所有　些点的　分都是　分 `1.0` 。

在此　　之外，　分　始衰　，衰　率由 `scale`　（此例中的　　`5` ）和 衰　　`decay`（此例中 `0.5` ）共同决定。　果是所有三个曲　在 `origin +/- (offset + scale)`　的　分都是 `0.5`，即点 `30` 和 `50` 。

`linear` 、 `exp` 和 `gauss`（　性、指数和高斯）函数三者之　的区　在于　　（ `origin +/- (offset + scale)` ）之外的曲　形状：

- linear 性函数是条直 ，一旦直 与横 0 相交，所有其他 的 分都是 0.0 。
- exp 指数函数是先 烈衰 然后 。
- gauss 高斯函数是 形的——它的衰 速率是先 慢，然后 快，最后又放 。

曲 的依据完全由期望 分 _score 的衰 速率来决定，即距原点 origin 的 。

回到我 的例子：用 希望租一个 敦市中心近（ { "lat": 51.50, "lon": 0.12} ）且 不超 £100 英 的度假屋，而且与距 相比，我 的用 格更 敏感， 可以写成：

```
GET /_search
{
  "query": {
    "function_score": {
      "functions": [
        {
          "gauss": {
            "location": { ①
              "origin": { "lat": 51.5, "lon": 0.12 },
              "offset": "2km",
              "scale":  "3km"
            }
          }
        },
        {
          "gauss": {
            "price": { ②
              "origin": "50", ③
              "offset": "50",
              "scale":  "20"
            }
          },
          "weight": 2 ④
        }
      ]
    }
  }
}
```

① location 字段以地理坐 点 geo_point 映射。

② price 字段是数 。

③ 参 理解 格 句，理解 origin 什 是 50 而不是 100 。

④ price 句是 location 句 重的 倍。

location 句可以 理解 ：

- 以 敦市中作 原点 origin 。
- 所有距原点 origin 2km 内的位置的 分是 1.0 。

- 距中心 5km （ offset + scale ）的位置的 分是 0.5 。

理解 price 格 句

price 句使用了一个小技巧：用 希望 £100 英 以下的度假屋，但是例子中的原点被 置成 £50 英 ， 格不能 ，但肯定是越低越好，所以 £0 到 £100 英 内的所有 格都 是比 好的。

如果我 将原点 origin 被 置成 £100 英 ，那 低于 £100 英 的度假屋的 分会 低，与其 不如将原点 origin 和偏移量 offset 同 置成 £50 英 ， 就能使只有在 格高于 £100 英 （ origin + offset ） 分才会 低。

> **TIP** weight 参数可以被用来 整 个 句的 献度， 重 weight 的 是 1.0 。 个 会先与 个句子的 分相乘，然后再通 score_mode 的 置方式合并。

脚本 分

最后，如果所有 function_score 内置的函数都无法 足 用 景，可以使用 script_score 函数自行 。

个例子，想将利 空 作 因子加入到相 度 分 算，在 中，利 空 和以下三点相 ：

- price 度假屋 的 格。
- 会 用 的 ——某些等 的用 可以在 房 高于某个 threshold 格的 候享受折扣 discount 。
- 用 享受折扣后， 的 房 的利 margin 。

算 个度假屋利 的算法如下：

```
if (price < threshold) {
  profit = price * margin
} else {
  profit = price * (1 - discount) * margin;
}
```

我 很可能不想用 利 作 分， 会弱化其他如地点、受 迎度和特性等因子的作用，而是将利 用目 利 target 的百分比来表示，高于 目 的利 空 会有一个正向 分（大于 1.0 ），低于目 的利 空 会有一个 向分数（小于 1.0 ）：

```
if (price < threshold) {
  profit = price * margin
} else {
  profit = price * (1 - discount) * margin
}
return profit / target
```

Elasticsearch 里使用 Groovy 作 的脚本 言，它与JavaScript很像，上面 个算法用 Groovy

脚本表示如下：

```
price  = doc['price'].value ①
margin = doc['margin'].value ①

if (price < threshold) { ②
  return price * margin / target
}
return price * (1 - discount) * margin / target ②
```

① price 和 margin 量可以分 从文 的 price 和 margin 字段提取。

② threshold 、 discount 和 target 是作 参数 params 入的。

最 我 将 script_score 函数与其他函数一起使用：

```
GET /_search
{
  "function_score": {
    "functions": [
      { ...location clause... }, ①
      { ...price clause... }, ①
      {
        "script_score": {
          "params": { ②
            "threshold": 80,
            "discount": 0.1,
            "target": 10
          },
          "script": "price  = doc['price'].value; margin = doc['margin'].value;
          if (price < threshold) { return price * margin / target };
          return price * (1 - discount) * margin / target;" ③
        }
      }
    ]
  }
}
```

① location 和 price 句在 衰 函数 中解 。

② 将 些 量作 参数 params ，我 可以 改 脚本无 重新 。

③ JSON 不能接受内嵌的 行符，脚本中的 行符可以用 \n 或 ; 符号替代。

 个 根据用 地点和 格的需求，返回用 最 意的文 ，同 也考 到我 于盈利的要求。

TIP

`script_score` 函数提供了巨大的 活性，可以通 脚本 文 里的所有字段、当前 分 `_score` 甚至 、逆向文 率和字段 度 的信息（参 see {ref}/modules-advanced-scripting.html[脚本 文本 分]）。

有人 使用脚本 性能会有影 ，如果 脚本 行 慢，可以有以下三 ：

- 尽可能多的提前 算各 信息并将 果存入 个文 中。
- Groovy 很快，但没 Java 快。可以将脚本用原生的 Java 脚本重新 。（参 {ref}/modules-scripting-native.html[原生 Java 脚本]）。
- 那些最佳 分的文 用脚本，使用 重新 分 中提到的 `rescore` 功能。

可 的相似度算法

在 一 相 度和 分之前，我 会以一个更高 的 束本章 的内容：可 的相似度算法（Pluggable Similarity Algorithms）。 Elasticsearch 将 用 分算法 作 相似度算法，它也能 支持其他的一些算法， 些算法可以参考 {ref}/index-modules-similarity.html#configuration[相似度模] 文 。

Okapi BM25

能与 TF/IDF 和向量空 模型 美的就是 *Okapi BM25* ，它被 是 当今最先 的 排序函数。 BM25 源自 概率相 模型（probabilistic relevance model） ，而不是向量空 模型，但 个算法也和 Lucene 的 用 分函数有很多共通之 。

BM25 同 使用 、逆向文 率以及字段 一化，但是 个因子的定 都有 微区 。与其 解 BM25 公式，倒不如将 注点放在 BM25 所能 来的 好 上。

和度

TF/IDF 和 BM25 同 使用 逆向文 率 来区分普通 （不重要）和非普通 （重要），同 （参 ）文 里的某个 出 次数越 繁，文 与 个 就越相 。

不幸的是，普通 随 可 ， 上一个普通 在同一个文 中大量出 的作用会由于 在 所有 文 中的大量出 而被抵消掉。

曾 有个 期，将 最 普通的 （或 停用 ，参 停用 ）从索引中移除被 是一 准 践，TF/IDF 正是在 背景下 生的。TF/IDF 没有考 上限的 ，因 高 停用 已 被移除了。

Elasticsearch 的 `standard` 准分析器（ `string` 字段 使用）不会移除停用 ，因 尽管 些 的重要性很低，但也不是 无用 。 致：在一个相当 的文 中，像 `the` 和 `and` 出 的数量会高得 ，以致它 的 重被人 放大。

一方面，BM25 有一个上限，文 里出 5 到 10 次的 会比那些只出 一 次的 相 度有着 著影 。但是如 TF/IDF 与 BM25 的 和度 所 ，文 中出 20 次的 几乎与那些出 上千次的 有着相同的影 。

就是 非 性 和度（*nonlinear term-frequency saturation*）。

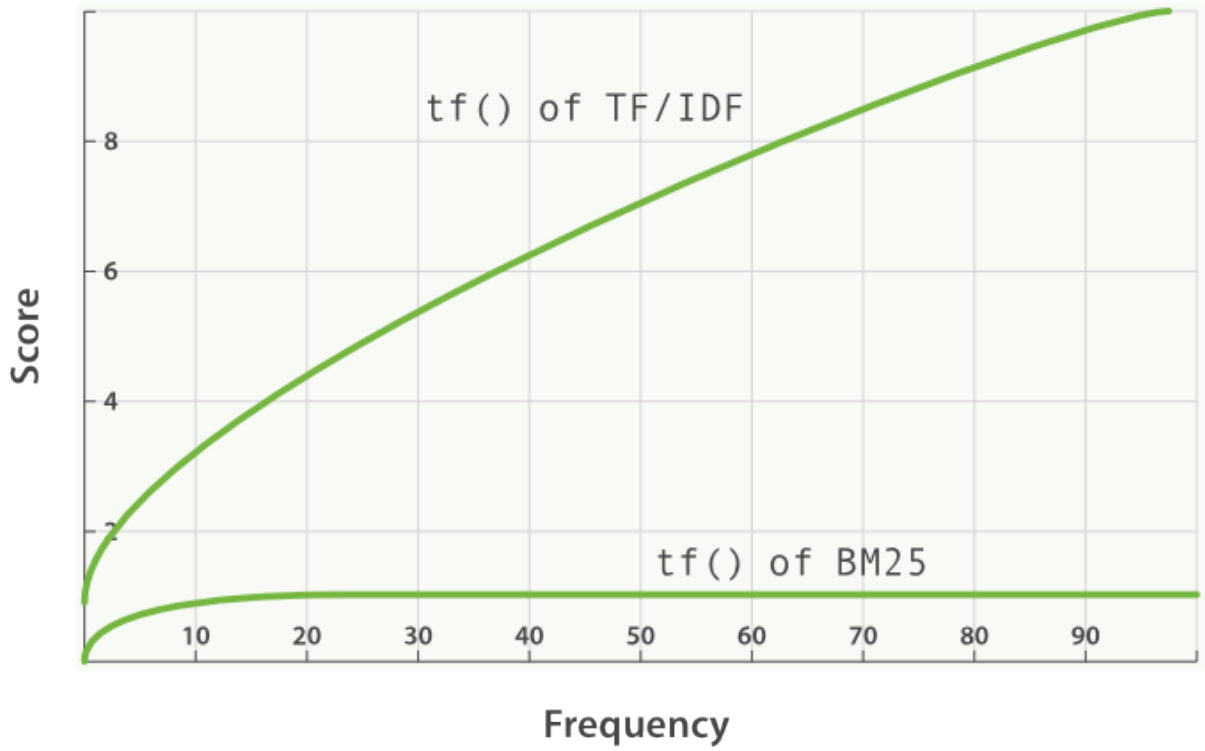

Figure 8. TF/IDF 与 BM25 的 和度

字段 度 一化（**Field-length normalization**）

在 字段 一化 中，我 提到 Lucene 会 短字段比 字段更重要：字段某个 的 度所 来的重要性会被 个字段 度抵消，但是 的 分函数会将所有字段以同等方式 待。它 所有 短的 `title` 字段比所有 的 `body` 字段更重要。

BM25 当然也 短字段 有更多的 重，但是它会分 考 个字段内容的平均 度，就能区分短 `title` 字段和 title 字段。

CAUTION	在 重提升 中，已 `title` 字段因 其 度比 `body` 字段 自然 有更高的 重提升 。由于字段 度的差 只能 用于 字段， 自然的 重提升会在使用 BM25 消失。

BM25

不像 TF/IDF，BM25 有一个比 好的特性就是它提供了 个可 参数：

`k1`
个参数控制着 果在 和度中的上升速度。 `1.2` 。 越小 和度 化越快， 越大 和度 化越慢。

`b`
个参数控制着字段 一 所起的作用， `0.0` 会禁用 一化， `1.0` 会 用完全 一化。 `0.75` 。

在 践中， BM25 是 外一回事， `k1` 和 `b` 的 用于 大多数文 集合，但最 是会因

文　集不同而有所区　，　了　到文　集合的最　　，就必　　参数　行反　修改　　。

更改相似度

相似度算法可以按字段指定，只需在映射中　不同字段　定即可：

```
PUT /my_index
{
  "mappings": {
    "doc": {
      "properties": {
        "title": {
          "type":       "string",
          "similarity": "BM25" ①
        },
        "body": {
          "type":       "string",
          "similarity": "default" ②
        }
      }
  }
}
```

① `title` 字段使用 BM25 相似度算法。

② `body` 字段用　　相似度算法（参　　用　分函数）。

目前，Elasticsearch　　不支持更改已有字段的相似度算法　　`similarity`　　映射，只能通数据重新建立索引来　到目的。

配置 BM25

配置相似度算法和配置分析器很相似，自定　相似度算法可以在　建索引　指定，例如：

```
PUT /my_index
{
  "settings": {
    "similarity": {
      "my_bm25": { ①
        "type": "BM25",
        "b":    0 ②
      }
    }
  },
  "mappings": {
    "doc": {
      "properties": {
        "title": {
          "type":       "string",
          "similarity": "my_bm25" ③
        },
        "body": {
          "type":       "string",
          "similarity": "BM25" ④
        }
      }
    }
  }
}
```

① 建一个基于内置 BM25，名 my_bm25 的自定 相似度算法。

② 禁用字段 度 化（field-length normalization）。参 BM25 。

③ title 字段使用自定 相似度算法 my_bm25 。

④ 字段 body 使用内置相似度算法 BM25 。

TIP 自定 的相似度算法可以通 索引，更新索引 置， 索引 个 程 行更新。 可以无 重建索引又能 不同的相似度算法配置。

相 度是最后 10% 要做的事情

本章介 了 Lucene 是如何基于 TF/IDF 生成 分的。理解 分 程是非常重要的，就可以根据具体的 分 果 行 、 、 弱和定制。

践中， 的 合就能提供很好的搜索 果，但是 了 得 具有成效 的搜索 果，就必 反推敲修改前面介 的 些 方法。

通常， 策略字段 用 重提升，或通 句 的 整来 某个句子的重要性 些方法，就足以 得良好的 果。有 ，如果 Lucene 基于 的 TF/IDF 模型不再 足 分需求（例如希望基于 或距 来 分）， 需要更具侵略性的 整。

除此之外，相 度的 就有如兔子洞，一旦跳 去就很 再出来。 最相 个概念是一个

以触及的模糊目 ，通常不同人 文 排序又有着不同的想法， 很容易使人陷入持 反 整而没有明 展的怪圈。

我 烈建 不要陷入 怪圈，而要 控 量搜索 果。 控用 点 最 端 果的 次， 可以是前 10 个文 ，也可以是第一 的；用 不 看首次搜索的 果而直接 行第二次 的 次；用 来回点 并 看搜索 果的 次，等等 如此 的信息。

些都是用来 搜索 果与用 之 相 程度的指 。如果 能返回高相 的文 ，用 会 前五中的一个，得到想要的 果，然后 。不相 的 果会 用 来回点 并 新的搜索条件。

一旦有了 些 控手段，想要 就并不 ， 作 整， 控用 的行 改 并做 当反 。本章介 的一些工具就只是工具而已，要想物尽其用并将搜索 果提高到 高的水平，唯一途径就是需要具 能 度量用 行 的 大能力。