

placeholder1

深入搜索

在 **基础** 中涵了基本工具并它有足的描述，我能够始用 Elasticsearch 搜索数据。用不了多，就会我想要的更多：希望匹配更活，排名果更精，不同域下搜索更具体。

想要，只知道如何使用 **match** 是不的，我需要理解数据以及如何能搜索到它。本章会解如何索引和我的数据我利用的相似度 (word proximity)、部分匹配 (partial matching)、模糊匹配 (fuzzy matching) 以及言感知 (language awareness) 些。

理解个如何献相度分 **_score** 有助于我的：保我 的最佳匹配文出在果首，以及削果中几乎不相的“尾 (long tail)”。

搜索不 是全文搜索：我很大一部分数据都是化的，如日期和数字。我会以明化搜索与全文搜索最高效的合方式始本章的内容。

化搜索

化搜索 (*Structured search*) 是指有探那些具有内在数据的程。比如日期、和数字都是化的：它有精的格式，我可以些格式行操作。比常的操作包括比数字或的，或判定个的大小。

文本也可以是化的。如彩色可以有散的色集合：**(red)**、**(green)**、**(blue)**。一个博客可能被了**分布式 (distributed)** 和**搜索 (search)**。商站上的商品都有UPCs (通用品 Universal Product Codes) 或其他的唯一，它都需要遵从格定的、化的格式。

在化中，我得到的果是非是即否，要存于集合之中，要存在集合之外。化不关心文件的相度或分；它的文包括或排除理。

在上是能通的，因一个数字不能比其他数字更合存于某个相同。果只能是：存于之中，抑或反之。同，于化文本来，一个要相等，要不等。没有更似概念。

精

当行精，我会使用器 (filters)。器很重要，因它行速度非常快，不会算相度 (直接跳了整个分段) 而且很容易被存。我会在本章后面的器存中器的性能，不在只要住：尽可能多的使用式。

term 数字

我首先来看最常用的 **term**，可以用它理数字 (numbers)、布 (Booleans)、日期 (dates) 以及文本 (text)。

我以下面的例子始介，建并索引一些表示品的文，文里有字段 `price` 和 `productID` ('格' 和 '品ID')：

```
POST /my_store/products/_bulk
{ "index": { "_id": 1 }}
{ "price" : 10, "productID" : "XHDK-A-1293-#fJ3" }
{ "index": { "_id": 2 }}
{ "price" : 20, "productID" : "KDKE-B-9947-#kL5" }
{ "index": { "_id": 3 }}
{ "price" : 30, "productID" : "JODL-X-1937-#pV7" }
{ "index": { "_id": 4 }}
{ "price" : 30, "productID" : "QQPX-R-3956-#aD8" }
```

我想要做的是具有某个格的所有品，有系数据背景的人肯定熟悉SQL，如果我将其用SQL形式表，会是下面：

```
SELECT document
FROM products
WHERE price = 20
```

在Elasticsearch的表式(query DSL)中，我可以使用term到相同的目的。term会是我指定的精。作其本身，term是的。它接受一个字段名以及我希望的数：

```
{
  "term" : {
    "price" : 20
  }
}
```

通常当一个精的候，我不希望行分算。只希望文行包括或排除的算，所以我使用constant_score以非分模式来行term并以一作一分。

最合的果是一个constant_score，它包含一个term：

```
GET /my_store/products/_search
{
  "query" : {
    "constant_score" : { ①
      "filter" : {
        "term" : { ②
          "price" : 20
        }
      }
    }
  }
}
```

① 我用constant_score将term化成器

② 我之前看到的 term

行后，一个所搜索到的结果与我期望的一致：只有文 2 命中并作 果返回（因 只有 2 的格是 20）：

```
"hits" : [
  {
    "_index" : "my_store",
    "_type" : "products",
    "_id" : "2",
    "_score" : 1.0, ①
    "_source" : {
      "price" : 20,
      "productID" : "KDKE-B-9947-#kL5"
    }
  }
]
```

① 置于 filter 句内不行 分或相 度的 算，所以所有的 果都会返回一个 分 1。

term 文本

如本部分始提到的一 ，使用 term 匹配字符串和匹配数字一 容易。如果我想要某个具体 UPC ID 的 品，使用 SQL 表 式会是如下：

```
SELECT product
FROM products
WHERE productID = "XHDK-A-1293-#fJ3"
```

成 表 式 (query DSL)，同 使用 term ，形式如下：

```
GET /my_store/products/_search
{
  "query" : {
    "constant_score" : {
      "filter" : {
        "term" : {
          "productID" : "XHDK-A-1293-#fJ3"
        }
      }
    }
  }
}
```

但 里有个小 ！我 无法 得期望的 果。什 ？ 不在 term ，而在于索引数据的方式。如果我 使用 analyze API (分析 API)，我 可以看到 里的 UPC 被拆分成多个更小的 token：

```
GET /my_store/_analyze
{
  "field": "productID",
  "text": "XHDK-A-1293-#fJ3"
}
```

```
{
  "tokens" : [ {
    "token" : "xhdk",
    "start_offset" : 0,
    "end_offset" : 4,
    "type" : "<ALPHANUM>",
    "position" : 1
  }, {
    "token" : "a",
    "start_offset" : 5,
    "end_offset" : 6,
    "type" : "<ALPHANUM>",
    "position" : 2
  }, {
    "token" : "1293",
    "start_offset" : 7,
    "end_offset" : 11,
    "type" : "<NUM>",
    "position" : 3
  }, {
    "token" : "fj3",
    "start_offset" : 13,
    "end_offset" : 16,
    "type" : "<ALPHANUM>",
    "position" : 4
  } ]
}
```

里有几点需要注意：

- Elasticsearch 用 4 个不同的 token 而不是 1 个 token 来表示 1 个 UPC。
- 所有字母都是小写的。
- 失了 # 字符和哈希符 (#)。

所以当我用 term 精 XHDK-A-1293-#fJ3 的时候，找不到任何文，因为它并不在我的倒排索引中，正如前面呈现出的分析结果，索引里有四个 token。

然 ID 或其他任何精的理方式并不是我想要的。

为了避免 ，我需要告诉 Elasticsearch 字段具有精，要将其置成 not_analyzed 无需分析的。我可以在 自定字段映射 中看它的用法。为了修正搜索结果，我需要首先删除旧索引（因它的映射不再正）然后建一个能正映射的新索引：

```

DELETE /my_store ①

PUT /my_store ②
{
  "mappings" : {
    "products" : {
      "properties" : {
        "productID" : {
          "type" : "string",
          "index" : "not_analyzed" ③
        }
      }
    }
  }
}

```

- ① 除索引是必 的，因 我 不能更新已存在的映射。
- ② 在索引被 除后，我 可以 建新的索引并 其指定自定 映射。
- ③ 里我 告 Elasticsearch，我 不想 productID 做任何分析。

在我 可以 文 重建索引：

```

POST /my_store/products/_bulk
{ "index": { "_id": 1 }}
{ "price" : 10, "productID" : "XHDK-A-1293-#fJ3" }
{ "index": { "_id": 2 }}
{ "price" : 20, "productID" : "KDKE-B-9947-#kL5" }
{ "index": { "_id": 3 }}
{ "price" : 30, "productID" : "JODL-X-1937-#pV7" }
{ "index": { "_id": 4 }}
{ "price" : 30, "productID" : "QQPX-R-3956-#aD8" }

```

此 ， term 就能搜索到我 想要的 果， 我 再次搜索新索引 的数据（注意， 和 并没有 生任何改 ， 改 的是数据映射的方式）：

```

GET /my_store/products/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": {
          "productID": "XHDK-A-1293-#fJ3"
        }
      }
    }
  }
}

```

因 `productID` 字段是未分析 的， `term` 不会 其做任何分析， 会 行精 并返回文 1 。成功！

内部 器的操作

在内部， Elasticsearch 会在 行非 分 的 行多个操作：

1. 匹配文 .

`term` 在倒排索引中 `XHDK-A-1293-#fJ3` 然后 取包含 `term` 的所有文 。本例中，只有文 1 足我 要求。

2. 建 `bitset`.

器会 建一个 `bitset` （一个包含 0 和 1 的数 ），它描述了 个文 会包含 `term` 。匹配文 的 志位是 1 。本例中，`bitset` 的 `[1,0,0,0]` 。在内部，它表示成一个 `"roaring bitmap"` ，可以同 稀疏或密集的集合 行高效 。

3. 迭代 `bitset(s)`

一旦 个 生成了 `bitsets` ， Elasticsearch 就会循 迭代 `bitsets` 从而 到 足所有 条件的匹配文 的集合。 行 序是 式的，但一般来 先迭代稀疏的 `bitset` （因 它可以排除掉大量的文 ）。

4. 量使用 数.

Elasticsearch 能 存非 分 从而 取更快的 ，但是它也会不太 明地 存一些使用 少的 西。非 分 算因 倒排索引已 足 快了，所以我 只想 存那些我 知道 在将来会被再次使用的 ，以避免 源的浪 。

了 以上 想， Elasticsearch 会 个索引跟踪保留 使用的 史状 。如果 在最近的 256 次 中会被用到，那 它就会被 存到内存中。当 `bitset` 被 存后， 存会在那些低于 10,000 个文 （或少于 3% 的 索引数）的段（segment）中被忽略。 些小的段即将会消失，所以 它 分配 存是一 浪 。

情况并非如此（ 行有它的 性， 取决于 是如何重新 的，有些 式的算法是基于 代 的），理 上非 分 先于 分 行。非 分 任 旨在降低那些将 分 算

来更高成本的文 数量，从而 到快速搜索的目的。

从概念上 住非 分 算是首先 行的， 将有助于写出高效又快速的搜索 求。

合 器

前面的 个例子都是 个 器 (filter) 的使用方式。 在 用中，我 很有可能会 多个 或字段。比方 ， 用 Elasticsearch 来表 下面的 SQL ?

```
SELECT product
FROM products
WHERE (price = 20 OR productID = "XHDK-A-1293-#fJ3")
AND (price != 30)
```

情况下，我 需要 **bool** (布) 器。 是个 合 器 (*compound filter*) ，它可以接受多个其他 器作 参数，并将 些 器 合成各式各 的布 () 合。

布 器

一个 **bool** 器由三部分 成：

```
{
  "bool" : {
    "must" : [],
    "should" : [],
    "must_not" : []
  }
}
```

must

所有的 句都必 (must) 匹配，与 AND 等 。

must_not

所有的 句都不能 (must not) 匹配，与 NOT 等 。

should

至少有一个 句要匹配，与 OR 等 。

就 ! 当我 需要多个 器 ，只 将它 置入 **bool** 器的不同部分即可。

NOTE 一个 **bool** 器的 个部分都是可 的 (例如，我 可以只有一个 **must** 句)，而且 个部分内部可以只有一个或一 器。

用 Elasticsearch 来表示本部分 始 的 SQL 例子，将 个 **term** 器置入 **bool** 器的 **should** 句内，再 加一个 句 理 NOT 非的条件：

```

GET /my_store/products/_search
{
  "query" : {
    "filtered" : { ①
      "filter" : {
        "bool" : {
          "should" : [
            { "term" : {"price" : 20}}, ②
            { "term" : {"productID" : "XHDK-A-1293-#fJ3"} } ②
          ],
          "must_not" : {
            "term" : {"price" : 30} ③
          }
        }
      }
    }
  }
}

```

① 注意，我 然需要一个 `filtered` 将所有的 西包起来。

② 在 `should` 句 里面的 个 `term` 器与 `bool` 器是父子 系， 个 `term` 条件需要匹配其一。

③ 如果一个 品的 格是 30， 那 它会自 被排除，因 它 于 `must_not` 句里面。

我 搜索的 果返回了 2 个命中 果， 个文 分 匹配了 `bool` 器其中的一个条件：

```

"hits" : [
  {
    "_id" :      "1",
    "_score" :  1.0,
    "_source" : {
      "price" :      10,
      "productID" : "XHDK-A-1293-#fJ3" ①
    }
  },
  {
    "_id" :      "2",
    "_score" :  1.0,
    "_source" : {
      "price" :      20, ②
      "productID" : "KDKE-B-9947-#kL5"
    }
  }
]

```

① 与 `term` 器中 `productID = "XHDK-A-1293-#fJ3"` 条件匹配

② 与 `term` 器中 `price = 20` 条件匹配

嵌套布 器

尽管 `bool` 是一个 合的 器，可以接受多个子 器，需要注意的是 `bool` 器本身 然 只是一个 器。 意味着我 可以将一个 `bool` 器置于其他 `bool` 器内部， 我 提供了 任意 布 行 理的能力。

于以下 个 SQL 句：

```
SELECT document
FROM products
WHERE productID      = "KDKE-B-9947-#kL5"
  OR (    productID = "JODL-X-1937-#pV7"
    AND price       = 30 )
```

我 将其 成一 嵌套的 `bool` 器：

```
GET /my_store/products/_search
{
  "query": {
    "filtered": {
      "filter": {
        "bool": {
          "should": [
            { "term": {"productID": "KDKE-B-9947-#kL5"}}, ①
            { "bool": { ①
              "must": [
                { "term": {"productID": "JODL-X-1937-#pV7"}}, ②
                { "term": {"price": 30}} ②
              ]
            }
          ]
        }
      }
    }
  }
}
```

① 因 `term` 和 `bool` 器是兄弟 系，他 都 于外 的布 `should` 的内部，返回的命中文 至少 匹配其中一个 器的条件。

② 一个 `term` 句作 兄弟 系，同 于 `must` 句之中，所以返回的命中文 要必 都能同 匹配 个条件。

得到的 果有 个文 ，它 各匹配 `should` 句中的一个条件：

```

"hits" : [
  {
    "_id" : "2",
    "_score" : 1.0,
    "_source" : {
      "price" : 20,
      "productID" : "KDKE-B-9947-#kL5" ①
    }
  },
  {
    "_id" : "3",
    "_score" : 1.0,
    "_source" : {
      "price" : 30, ②
      "productID" : "JODL-X-1937-#pV7" ②
    }
  }
]

```

① 一个 `productID` 与外层的 `bool` 器 `should` 里的唯一一个 `term` 匹配。

② 一个字段与嵌套的 `bool` 器 `must` 里的一个 `term` 匹配。

只是个例子，但足以展示布尔器可以用来作构造条件的基本建模。

多个精确匹配

`term` 用于一个非常有用，但通常我可能想搜索多个。如果我想要价格字段 \$20 或 \$30 的文章如何理？

不需要使用多个 `term`，我只要用一个 `terms`（注意末尾的 `s`），`terms` 好比是 `term` 的复数形式（以英文名的复数做比）。

它几乎与 `term` 的使用方式一模一样，与指定一个格不同，我只要将 `term` 字段的改数值即可：

```

{
  "terms" : {
    "price" : [20, 30]
  }
}

```

与 `term` 一样，也需要将其置入 `filter` 句的常量分段中使用：

```

GET /my_store/products/_search
{
  "query" : {
    "constant_score" : {
      "filter" : {
        "terms" : { ①
          "price" : [20, 30]
        }
      }
    }
  }
}

```

① ↑ terms 被置于 constant_score 中

结果返回第二、第三和第四个文档：

```

"hits" : [
  {
    "_id" : "2",
    "_score" : 1.0,
    "_source" : {
      "price" : 20,
      "productID" : "KDKE-B-9947-#kL5"
    }
  },
  {
    "_id" : "3",
    "_score" : 1.0,
    "_source" : {
      "price" : 30,
      "productID" : "JODL-X-1937-#pV7"
    }
  },
  {
    "_id": "4",
    "_score": 1.0,
    "_source": {
      "price": 30,
      "productID": "QQPX-R-3956-#aD8"
    }
  }
]

```

包含，而不是相等

一定要了解 term 和 terms 是包含 (contains) 操作，而非 等 (equals) (判断)。如何理解 句？

如果我 有一个 term () 器 { "term" : { "tags" : "search" } } , 它会与以下 个文 同匹配：

```
{ "tags" : ["search"] }
{ "tags" : ["search", "open_source"] } ①
```

① 尽管第二个文 包含除 search 以外的其他 , 它 是被匹配并作 果返回。

回 一下 term 是如何工作的？ Elasticsearch 会在倒排索引中 包括某 term 的所有文 , 然后造一个 bitset 。在我 的例子中，倒排索引表如下：

Token	DocIDs
open_source	2
search	1,2

当 term 匹配 search , 它直接在倒排索引中 到 并 取相 的文 ID, 如倒排索引所示, 里文 1 和文 2 均包含 , 所以 个文 会同 作 果返回。

NOTE

由于倒排索引表自身的特性，整个字段是否相等会 以 算，如果 定某个特定文 是否只 (only) 包含我 想要 的 ?首先我 需要在倒排索引中 到相 的 并 取文 ID, 然后再 描倒排索引中的 行 , 看它 是否包含其他的 terms 。

可以想象， 不 低效，而且代 高昂。正因如此， term 和 terms 是 必 包含 (must contain) 操作，而不是 必 精 相等 (must equal exactly) 。

精 相等

如果定期望得到我 前面 的那 行 (即整个字段完全相等) , 最好的方式是 加并索引 一个字段, 个字段用以存 字段包含 的数量, 同 以上面提到的 个文 例, 在我 包括了一个数的新字段 :

```
{ "tags" : ["search"], "tag_count" : 1 }
{ "tags" : ["search", "open_source"], "tag_count" : 2 }
```

一旦 加 个用来索引 term 数目信息的字段, 我 就可以 造一个 constant_score , 来 保 果中的文 所包含的 数量与要求是一致的 :

```

GET /my_index/my_type/_search
{
  "query": {
    "constant_score" : {
      "filter" : {
        "bool" : {
          "must" : [
            { "term" : { "tags" : "search" } }, ①
            { "term" : { "tag_count" : 1 } } ②
          ]
        }
      }
    }
  }
}

```

① 所有包含 term `search` 的文 。

② 保文 只有一个 。

个 在只会匹配具有 个 `search` 的文 , 而不是任意一个包含 `search` 的文 。

本章到目前 止, 于数字, 只介 如何 理精 。 上, 数字 行 有 会更有用。例
如, 我 可能想要 所有 格大于 \$20 且小于 \$40 美元的 品。

在 SQL 中, 可以表示 :

```

SELECT document
FROM products
WHERE price BETWEEN 20 AND 40

```

Elasticsearch 有 `range` , 不出所料地, 可以用它来 于某个 内的文 :

```

"range" : {
  "price" : {
    "gte" : 20,
    "lte" : 40
  }
}

```

`range` 可同 提供包含 (inclusive) 和不包含 (exclusive) 表 式, 可供 合的 如下:

- `gt:>` 大于 (greater than)
- `lt:<` 小于 (less than)
- `gte:>=` 大于或等于 (greater than or equal to)

- **lte:** 小于或等于 (less than or equal to)

下面是一个 **range** 的例子：

```
GET /my_store/products/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "range": {
          "price": {
            "gte": 20,
            "lt": 40
          }
        }
      }
    }
  }
}
```

如果想要 无界 (比方 >20) , 只 省略其中一 的限制 :

```
"range": {
  "price": {
    "gt": 20
  }
}
```

日期

range 同 可以 用在日期字段上 :

```
"range": {
  "timestamp": {
    "gt": "2014-01-01 00:00:00",
    "lt": "2014-01-07 00:00:00"
  }
}
```

当使用它 理日期字段 , **range** 支持 日期 算 (date math) 行操作, 比方 , 如果我想 在 去一小 内的所有文 :

```
"range" : {  
    "timestamp" : {  
        "gt" : "now-1h"  
    }  
}
```

个 器会一直 在 去一个 小 内的所有文 , 器作 一个 滑 口 (*sliding window*) 来 文 。

日期 算 可以被 用到某个具体的 , 并非只能是一个像 now 的占位符。只要在某个日期后加上一个双管符号 (||) 并 跟一个日期数学表 式就能做到 :

```
"range" : {  
    "timestamp" : {  
        "gt" : "2014-01-01 00:00:00",  
        "lt" : "2014-01-01 00:00:00||+1M" ①  
    }  
}
```

① 早于 2014 年 1 月 1 日加 1 月 (2014 年 2 月 1 日 零)

日期 算是 日 相 (*calendar aware*) 的, 所以它不 知道 月的具体天数, 知道某年的 天数 (年) 等信息。更 的内容可以参考 : {ref}/mapping-date-format.html[格式参考文]。

字符串

range 同 可以 理字符串字段, 字符串 可采用 字典 序 (*lexicographically*) 或字母序 (*alphabetically*) 。例如, 下面 些字符串是采用字典序 (*lexicographically*) 排序的 :

- 5, 50, 6, B, C, a, ab, abb, abc, b

NOTE

在倒排索引中的 就是采取字典 序 (*lexicographically*) 排列的, 也是字符串 可以使用 个 序来 定的原因。

如果我 想 从 a 到 b (不包含) 的字符串, 同 可以使用 **range** 法 :

```
"range" : {  
    "title" : {  
        "gte" : "a",  
        "lt" : "b"  
    }  
}
```

注意基数

数字和日期字段的索引方式使高效地， Elasticsearch 上是在内的一个都行 term 器，会比日期或数字的慢多。字符串 在低基数 (low cardinality) 字段（即只有少量唯一）可以正常工作，但是唯一越多，字符串的算会越慢。

理 Null

回想在之前例子中，有的文有名 tags () 的字段，它是个多字段，一个文可能有一个或多个，也可能根本就没有。如果一个字段没有，那如何将它存入倒排索引中的？

是个有欺性的，因答案是：什都不存。我看看之前内容里提到的倒排索引：

Token	DocIDs
open_source	2
search	1,2

如何将某个不存在的字段存 在个数据中？无法做到！的，一个倒排索引只是一个 token 列表和与之相的文信息，如果字段不存在，那它也不会持有任何 token，也就无法在倒排索引中表。

最，也就意味着，`null, []` (空数) 和 `[null]` 所有都是等的，它无法存于倒排索引中。

然，世界并不，数据往往会有失字段，或有式的空或空数。了这些状况， Elasticsearch 提供了一些工具来理空或失。

存在

第一件武器就是 `exists` 存在。一个会返回那些在指定字段有任何的文，我 索引一些示例文 并用的例子来 明：

```
POST /my_index/posts/_bulk
{ "index": { "_id": "1" } }
{ "tags" : ["search"] } ①
{ "index": { "_id": "2" } }
{ "tags" : ["search", "open_source"] } ②
{ "index": { "_id": "3" } }
{ "other_field" : "some data" } ③
{ "index": { "_id": "4" } }
{ "tags" : null } ④
{ "index": { "_id": "5" } }
{ "tags" : ["search", null] } ⑤
```

- ① tags 字段有 1 个。
- ② tags 字段有 2 个。
- ③ tags 字段 失。
- ④ tags 字段被置 null。
- ⑤ tags 字段有 1 个 和 1 个 null。

以上文 集合中 tags 字段 的倒排索引如下：

Token	DocIDs
open_source	2
search	1,2,5

我 的目 是 到那些被 置 字段的文 , 并不 心 的具体内容。只要它存在于文 中即可, 用 SQL 的 就是用 IS NOT NULL 非空 行 :

```
SELECT tags
FROM posts
WHERE tags IS NOT NULL
```

在 Elasticsearch 中, 使用 exists 的方式如下:

```
GET /my_index/posts/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "exists": { "field": "tags" }
      }
    }
  }
}
```

个 返回 3 个文 :

```

"hits" : [
  {
    "_id" : "1",
    "_score" : 1.0,
    "_source" : { "tags" : ["search"] }
  },
  {
    "_id" : "5",
    "_score" : 1.0,
    "_source" : { "tags" : ["search", null] } ①
  },
  {
    "_id" : "2",
    "_score" : 1.0,
    "_source" : { "tags" : ["search", "open source"] }
  }
]

```

① 尽管文 5 有 `null`，但它会被命中返回。字段之所以存在，是因为（`search`）可以被索引，所以 `null` 不会生任何影。

而易，只要 `tags` 字段存在（term）的文 都会命中并作 果返回，只有 3 和 4 个文 被排除。

失

个 `missing` 本 上与 `exists` 恰好相反：它返回某个特定 无 字段的文，与以下 SQL 表 的意思似：

```

SELECT tags
FROM posts
WHERE tags IS NULL

```

我 将前面例子中 `exists` 成 `missing`：

```

GET /my_index/posts/_search
{
  "query" : {
    "constant_score" : {
      "filter": {
        "missing" : { "field" : "tags" }
      }
    }
  }
}

```

按照期望的那，我 得到 3 和 4 个文（一个文 的 `tags` 字段没有）：

```

"hits" : [
  {
    "_id" : "3",
    "_score" : 1.0,
    "_source" : { "other_field" : "some data" }
  },
  {
    "_id" : "4",
    "_score" : 1.0,
    "_source" : { "tags" : null }
  }
]

```

当 `null` 的意思是 `null`

有 时候我 需要区分一个字段是没有 , 是它已被 式的 置成了 `null` 。在之前例子中, 我 看到的 行 是无法做到 点的; 数据被 失了。不 幸 的是, 我 可以 将 式的 `null` 替 成我 指定占位符 (*placeholder*) 。

在 字符串 (string) 、数字 (numeric) 、布 (Boolean) 或日期 (date) 字段指定映射 , 同 可以 之 置 `null_value` 空 , 用以 理 式 `null` 的情况。不 即使如此, 是会将一个没有 的字段从倒排索引中排除。

当 合 的 `null_value` 空 的 候, 需要保 以下几点 :

- 它会匹配字段的 型, 我 不能 一个 `date` 日期字段 置字符串 型的 `null_value` 。
- 它必 与普通 不一 , 可以避免把 当成 `null` 空的情况。

象上的存在与 失

不 可以 核心 型, `exists` and `missing` 可以 理一个 象的内部字段。以下面文 例 :

```
{
  "name" : {
    "first" : "John",
    "last" : "Smith"
  }
}
```

我 不 可以 `name.first` 和 `name.last` 的存在性, 也可以 `name` , 不 在 映射 中, 如上 象的内部是个扁平的字段与 (field-value) 的 , 似下面 :

```
{  
    "name.first" : "John",  
    "name.last" : "Smith"  
}
```

那我如何用 `exists` 或 `missing` 检查 `name` 字段？`name` 字段并不真存在于倒排索引中。

原因是当我运行下面一个的时候：

```
{  
    "exists" : { "field" : "name" }  
}
```

行的是：

```
{  
    "bool": {  
        "should": [  
            { "exists": { "field": "name.first" }},  
            { "exists": { "field": "name.last" }}  
        ]  
    }  
}
```

也就意味着，如果 `first` 和 `last` 都是空，那么 `name` 个命名空间才会被不存在。

于存

在本章前面（[器的内部操作](#)）中，我已经介绍了器是如何算的。其核心是采用一个 `bitset` 与器匹配的文。Elasticsearch 地把些 `bitset` 存起来以随后使用。一旦存成功，`bitset` 可以用任何已使用的相同器，而无需再次算整个器。

些 `bitsets` 存是“智能”的：它以量方式更新。当我索引新文，只需将那些新文加入已有 `bitset`，而不是整个存一遍又一遍的重算。和系其他部分一，器是的，我无需担心存期。

独立的器存

属于一个件的 `bitsets` 是独立于它所属搜索求其他部分的。就意味着，一旦被存，一个可以被用作多个搜索求。`bitsets` 并不依于它所存在的上下文。使得存可以加速中常使用的部分，从而降低少、易的部分所来的消耗。

同，如果一个求重用相同的非分，它存的 `bitset` 可以被个搜索里的所有例所重用。

我看看下面例子中的，它足以以下任意一个条件的子件：

- 在收件箱中，且没有被的

- 不在收件箱中，但被注重要的

```
GET /inbox/emails/_search
{
  "query": {
    "constant_score": {
      "filter": {
        "bool": {
          "should": [
            { "bool": {
                "must": [
                  { "term": { "folder": "inbox" }}, ①
                  { "term": { "read": false }}
                ]
              }},
            { "bool": {
                "must_not": {
                  "term": { "folder": "inbox" } ①
                },
                "must": {
                  "term": { "important": true }
                }
              }}
            ]
          }
        }
      }
    }
}
```

① 两个过滤器是相同的，所以会使用同一 bitset。

尽管其中一个收件箱的条件是 `must` 句，一个是 `must_not` 句，但两者是完全相同的。意味着在第一个句行后，bitset 就会被算然后存起来供一个使用。当再次行一个，收件箱的两个过滤器已被存了，所以两个句都会使用已存的 bitset。

点与表式 (query DSL) 的可合性合得很好。它易被移到表达式的任何地方，或者在同一中的多个位置用。不能方便者，而且提升性能有直接的益。

自存行

在 Elasticsearch 的早版本中，的行是存一切可以存的形象。也通常意味着系存 bitsets 太富侵略性，从而因清理存来性能力。不如此，尽管很多过滤器都很容易被，但本上是慢于存的（以及从存中用）。存些过滤器的意不大，因可以地再次行过滤器。

一个倒排是非常快的，然后大多数文件却很少使用它。例如 `term` 字段 `"user_id"`：如果有上百万的用，个具体的用 ID 出的概率都很小。那一个过滤器存 bitsets 就不是很合算，因存的结果很可能在重用之前就被除了。

存的性能有着重的影。更重的是，它者以区分有良好表的存以及无用存

。解决这个问题，Elasticsearch 会基于使用次自存。如果一个非分在最近的 256 中被使用（次数取决于类型），那个就会作存的候。但是，并不是所有的片段都能保存 bitset。只有那些文数量超 10,000（或超文数量的 3%）才会存 bitset。因小的片段可以很快的行搜索和合并，里存的意不大。

一旦存了，非分算的 bitset 会一直留在存中直到它被除。除是基于 LRU 的：一旦存了，最近最少使用的器会被除。

全文搜索

我已介了搜索化数据的用示例，在来探全文搜索（full-text search）：

在全文字段中搜索到最相的文。

全文搜索个最重要的方面是：

相性 (Relevance)

它是与其果的相程度，并根据相程度果排名的一能力，算方式可以是 TF/IDF 方法（参 [相性的介](#)）、地理位置近、模糊相似，或其他的某些算法。

分析 (Analysis)

它是将文本有区的、化的 token 的一个程，（参 [分析的介](#)）目的是为了（a）建倒排索引以及（b）倒排索引。

一旦相性或分析个方面的，我所的境是于的而不是。

基于与基于全文

所有会或多或少的行相度算，但不是所有都有分析段。和一些特殊的完全不会文本行操作的（如 `bool` 或 `function_score`）不同，文本可以分成大家族：

基于的

如 `term` 或 `fuzzy` 的底不需要分析段，它一个行操作。用 `term` `Foo` 只要在倒排索引中准，并且用 TF/IDF 算法一个包含的文算相度分 `_score`。

住 `term` 只倒排索引的精匹配，点很重要，它不会的多性行理（如，`foo` 或 `FOO`）。里，无考是如何存入索引的。如果是将 `["Foo", "Bar"]` 索引存入一个不分析的（`not_analyzed`）包含精的字段，或者将 `Foo Bar` 索引到一个有 `whitespace` 空格分析器的字段，者的果都会是在倒排索引中有 `Foo` 和 `Bar` 个。

基于全文的

像 `match` 或 `query_string` 的是高，它了解字段映射的信息：

- 如果日期（`date`）或整数（`integer`）字段，它会将字符串分作日期或整数待。
- 如果一个（`not_analyzed`）未分析的精字符串字段，它会将整个字符串作一个待。
- 但如果要一个（`analyzed`）已分析的全文字段，它会先将字符串到一个合

的分析器，然后生成一个供 的 列表。

一旦 成了 列表， 个 会 个 逐一 行底 的 ， 再将 果合并， 然后 个文 生成 一个最 的相 度 分。

我 将会在随后章 中 个 程。

我 很少直接使用基于 的搜索，通常情况下都是 全文 行 ， 而非 个 ， 只需要 的 行 一个高 全文 （ 而在高 内部会以基于 的底 完成搜索）。

当我 想要 一个具有精 的 `not_analyzed` 未分析字段之前， 需要考虑， 是否真的采用 分 ， 或者非 分 会更好。

通常可以用是、非 二元 表示， 所以更 合用 ， 而且 做可以有效 利用 存：

```
NOTE
GET /_search
{
  "query": {
    "constant_score": {
      "filter": {
        "term": { "gender": "female" }
      }
    }
  }
}
```

匹配

匹配 `match` 是个 核心 。无 需要 什 字段， `match` 都 会是首 的 方式。它是一个高 全文 ， 表示它既能 理全文字段， 又能 理精 字段。

就是 ， `match` 主要的 用 景就是 行全文搜索， 我 以下面一个 例子来 明全文搜索是如何工作的：

索引一些数据

首先，我 使用 `bulk API` 建一些新的文 和索引：

```

DELETE /my_index ①

PUT /my_index
{ "settings": { "number_of_shards": 1 }} ②

POST /my_index/my_type/_bulk
{ "index": { "_id": 1 }}
{ "title": "The quick brown fox" }
{ "index": { "_id": 2 }}
{ "title": "The quick brown fox jumps over the lazy dog" }
{ "index": { "_id": 3 }}
{ "title": "The quick brown fox jumps over the quick dog" }
{ "index": { "_id": 4 }}
{ "title": "Brown fox brown dog" }

```

① 除已有的索引。

② 后，我 会在 [被破坏的相 性！](#) 中解 只 一个索引分配一个主分片的原因。

↑

我 用第一个示例来解 使用 `match` 搜索全文字段中的 个 ：

```

GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": "QUICK!"
    }
  }
}

```

Elasticsearch 行上面 个 `match` 的 是：

1. 字段 型。

`title` 字段是一个 `string` 型 (`analyzed`) 已分析的全文字段， 意味着 字符串本身也 被分析。

2. 分析 字符串。

将 的字符串 `QUICK!` 入 准分析器中， 出的 果是 个 `quick`。因 只有一个 ， 所以 `match` 行的是 个底 `term`。

3. 匹配文 。

用 `term` 在倒排索引中 `quick` 然后 取一 包含 的文 ， 本例的 果是文 : 1、2 和 3。

4. 个文 分。

用 term 算个文相度分 `_score`，是将 (term frequency, 即 `quick` 在相文的 `title` 字段中出的率) 和反向文率 (inverse document frequency, 即 `quick` 在所有文的 `title` 字段中出的率)，以及字段的度 (即字段越短相度越高) 相合的算方式。参 [相性的介](#)。

个程我以下()果：

```
"hits": [
  {
    "_id": "1",
    "_score": 0.5, ①
    "_source": {
      "title": "The quick brown fox"
    }
  },
  {
    "_id": "3",
    "_score": 0.44194174, ②
    "_source": {
      "title": "The quick brown fox jumps over the quick dog"
    }
  },
  {
    "_id": "2",
    "_score": 0.3125, ②
    "_source": {
      "title": "The quick brown fox jumps over the lazy dog"
    }
  }
]
```

①文 1 最相，因它的 `title` 字段更短，即 `quick` 占据内容的一大部分。

②文 3 比文 2 更具相性，因在文 2 中 `quick` 出了次。

多

如果我一次只能搜索一个，那全文搜索就会不太活，幸的是 `match` 多得：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": "BROWN DOG!"
    }
  }
}
```

上面 个 返回所有四个文 :

```
{  
  "hits": [  
    {  
      "_id": "4",  
      "_score": 0.73185337, ①  
      "_source": {  
        "title": "Brown fox brown dog"  
      }  
    },  
    {  
      "_id": "2",  
      "_score": 0.47486103, ②  
      "_source": {  
        "title": "The quick brown fox jumps over the lazy dog"  
      }  
    },  
    {  
      "_id": "3",  
      "_score": 0.47486103, ②  
      "_source": {  
        "title": "The quick brown fox jumps over the quick dog"  
      }  
    },  
    {  
      "_id": "1",  
      "_score": 0.11914785, ③  
      "_source": {  
        "title": "The quick brown fox"  
      }  
    }  
  ]  
}
```

① 文 4 最相 , 因 它包含 "brown" 次以及 "dog" 一次。

② 文 2、3 同 包含 brown 和 dog 各一次, 而且它 title 字段的 度相同, 所以具有相同的 分。

③ 文 1 也能匹配, 尽管它只有 brown 没有 dog 。

因 match 必 个 (["brown", "dog"]), 它在内部 上先 行 次 term , 然后将 次 的 果合并作 最 果 出。 了做到 点, 它将 个 term 包入一个 bool 中, 信息 布 。

以上示例告 我 一个重要信息: 即任何文 只要 title 字段里包含 指定 中的至少一个 就能匹配, 被匹配的 越多, 文 就越相 。

提高精度

用 任意 匹配文 可能会 致 果中出 不相 的 尾。 是 散 式搜索。可能我

只想搜索包含 所有 的文 , 也就是 , 不去匹配 brown OR dog , 而通 匹配 brown AND dog 到所有文 。

match 可以接受 operator 操作符作 入参数, 情况下 操作符是 or 。我 可以将它修改成 and 所有指定 都必 匹配 :

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": {①
        "query": "BROWN DOG!",
        "operator": "and"
      }
    }
  }
}
```

① match 的 需要做 整才能使用 operator 操作符参数。

个 可以把文 1 排除在外, 因 它只包含 个 中的一个。

控制精度

在 所有 与 任意 二 一有点 于非 即白。如果用 定 5 个 , 想 只包含其中 4 个的文 , 如何 理? 将 operator 操作符参数 置成 and 只会将此文 排除。

有 候 正是我 期望的, 但在全文搜索的大多数 用 景下, 我 既想包含那些可能相 的文 , 同 又排除那些不太相 的。 句 , 我 想要 于中 某 果。

match 支持 minimum_should_match 最小匹配参数, 我 可以指定必 匹配的 数用来表示一个文 是否相 。我 可以将其 置 某个具体数字, 更常用的做法是将其 置 一个百分数 , 因 我 无法控制用 搜索 入的 数量 :

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": {
        "query": "quick brown dog",
        "minimum_should_match": "75%"
      }
    }
  }
}
```

当 定百分比的 候, minimum_should_match 会做合 的事情 : 在之前三 的示例中, 75% 会自 被截断成 66.6% , 即三个里面 个 。无 个 置成什 , 至少包含一个 的文 才会被 是匹配的。

NOTE 参数 `minimum_should_match` 的置非常活，可以根据用入的数目用不同的。完整的信息参考文 {ref}/query-dsl-minimum-should-match.html#query-dsl-minimum-should-match

了完全理解 `match` 是如何理多的，我就需要看如何使用 `bool` 将多个条件合在一起。

合

在 合器中，我如何使用 `bool` 器通 `and`、`or` 和 `not` 合将多个器行合。在 中，`bool` 有似功能，只有一个重要的区。

器做二元判断：文是否出在果中？但更精妙，它除了决定一个文是否被包括在果中，会算文的相程度。

与器一，`bool` 也可以接受 `must`、`must_not` 和 `should` 参数下的多个句。比如：

```
GET /my_index/my_type/_search
{
  "query": {
    "bool": {
      "must": { "match": { "title": "quick" } },
      "must_not": { "match": { "title": "lazy" } },
      "should": [
        { "match": { "title": "brown" } },
        { "match": { "title": "dog" } }
      ]
    }
  }
}
```

以上的果返回 `title` 字段包含 `quick` 但不包含 `lazy` 的任意文。目前止，与 `bool`器的工作方式非常相似。

区就在于个 `should` 句，也就是：一个文不必包含 `brown` 或 `dog` 个，但如果一旦包含，我就它更相：

```
{
  "hits": [
    {
      "_id": "3",
      "_score": 0.70134366, ①
      "_source": {
        "title": "The quick brown fox jumps over the quick dog"
      }
    },
    {
      "_id": "1",
      "_score": 0.3312608,
      "_source": {
        "title": "The quick brown fox"
      }
    }
  ]
}
```

① 文 3 会比文 1 有更高 分是因 它同 包含 brown 和 dog 。

分 算

`bool` 会 个文 算相 度 分 `_score` , 再将所有匹配的 `must` 和 `should` 句的分数 `_score` 求和, 最后除以 `must` 和 `should` 句的 数。

`must_not` 句不会影 分 ; 它的作用只是将不相 的文 排除。

控制精度

所有 `must` 句必 匹配, 所有 `must_not` 句都必 不匹配, 但有多少 `should` 句 匹配 ? 情况下, 没有 `should` 句是必 匹配的, 只有一个例外 : 那就是当没有 `must` 句的 时候, 至少有一个 `should` 句必 匹配。

就像我 能控制 `match` 的精度 一 , 我 可以通 `minimum_should_match` 参数控制需要匹配的 `should` 句的数量, 它既可以是一个 的数字, 又可以是个百分比 :

```

GET /my_index/my_type/_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "brown" }},
        { "match": { "title": "fox" }},
        { "match": { "title": "dog" }}
      ],
      "minimum_should_match": 2 ①
    }
  }
}

```

① 也可以用百分比表示。

如果会将所有 足以下条件的文 返回： `title` 字段包含 "brown" AND "fox" 、 "brown" AND "dog" 或 "fox" AND "dog" 。如果有文 包含所有三个条件，它会比只包含 个的文 更相 。

如何使用布 匹配

目前 止，可能已 意 到多 `match` 只是 地将生成的 `term` 包 在一个 `bool` 中。如果使用 的 `or` 操作符， 个 `term` 都被当作 `should` 句， 就要求必至少匹配一条 句。以下 个 是等 的：

```

{
  "match": { "title": "brown fox"}
}

```

```

{
  "bool": {
    "should": [
      { "term": { "title": "brown" }},
      { "term": { "title": "fox" }}
    ]
  }
}

```

如果使用 `and` 操作符，所有的 `term` 都被当作 `must` 句，所以 所有 (all) 句都必 匹配。以下 个 是等 的：

```
{  
  "match": {  
    "title": {  
      "query": "brown fox",  
      "operator": "and"  
    }  
  }  
}
```

```
{  
  "bool": {  
    "must": [  
      { "term": { "title": "brown" }},  
      { "term": { "title": "fox" }}  
    ]  
  }  
}
```

如果指定参数 `minimum_should_match`，它可以通 `bool` 直接 ，使以下 个 等：

```
{  
  "match": {  
    "title": {  
      "query": "quick brown fox",  
      "minimum_should_match": "75%"  
    }  
  }  
}
```

```
{  
  "bool": {  
    "should": [  
      { "term": { "title": "brown" }},  
      { "term": { "title": "fox" }},  
      { "term": { "title": "quick" }}  
    ],  
    "minimum_should_match": 2 ①  
  }  
}
```

① 因 只有三条 句， `match` 的参数 `minimum_should_match` 75% 会被截断成 2 。即三条 `should` 句中至少有 条必 匹配。

当然，我 通常将 些 用 `match` 来表示，但是如果了解 `match` 内部的工作原理，我 就能根据自己的需要来控制 程。有些 候 个 `match` 无法 足需求，比如 某些 条件分配更高的 重。我 会在下一小 中看到 个例子。

句提升 重

当然 `bool` 不限于合的个 `match`，它可以合任意其他的，以及其他 `bool`。普遍的用法是通多个独立的分数，从而到一个文微其相度分 `_score` 的目的。

假想要于“full-text search（全文搜索）”的文，但我希望提及“Elasticsearch”或“Lucene”的文予更高的重，里更高重是指如果文出“Elasticsearch”或“Lucene”，它会比没有的出些的文得更高的相度分 `_score`，也就是，它会出在果集的更上面。

一个的 `bool` 允我写出如下非常 的：

```
GET /_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "content": { ①
            "query": "full text search",
            "operator": "and"
          }
        }
      },
      "should": [ ②
        { "match": { "content": "Elasticsearch" } },
        { "match": { "content": "Lucene" } }
      ]
    }
  }
}
```

① `content` 字段必包含 `full`、`text` 和 `search` 所有三个。

② 如果 `content` 字段也包含 `Elasticsearch` 或 `Lucene`，文会得更高的分 `_score`。

`should` 句匹配得越多表示文的相度越高。目前止挺好。

但是如果我想包含 `Lucene` 的有更高的重，并且包含 `Elasticsearch` 的句比 `Lucene` 的重更高，如何理？

我可以通指定 `boost` 来控制任何句的相的重，`boost` 的 1，大于 1 会提升一个句的相重。所以下面重写之前的：

```

GET /_search
{
  "query": {
    "bool": {
      "must": {
        "match": { ①
          "content": {
            "query": "full text search",
            "operator": "and"
          }
        }
      },
      "should": [
        { "match": {
          "content": {
            "query": "Elasticsearch",
            "boost": 3 ②
          }
        }},
        { "match": {
          "content": {
            "query": "Lucene",
            "boost": 2 ③
          }
        }}
      ]
    }
  }
}

```

① 些 句使用 的 `boost 1`。

② 条 句更 重要，因 它有最高的 `boost`。

③ 条 句比使用 的更重要，但它的的重要性不及 `Elasticsearch` 句。

`boost` 参数被用来提升一个 句的相 重（`boost` 大于 1）或降低相 重（`boost` 于 0 到 1 之），但是 提升或降低并不是 性的， 句 ，如果一个 `boost` 2，并不能 得 倍的 分 `_score`。

NOTE 相反，新的 分 `score` 会在 用 重提升之后被 _ 一化， 型的 都有自己的 一算法， 超出了本 的 ，所以不作介 。 的 ，更高的 `boost` 我 来更高的 分 `_score`。

如果不基于 TF/IDF 要 自己的 分模型，我 就需要 重提升的 程能有更多控制，可以使用 `function_score` 操 一个文 的 重提升方式而跳 一化 一 。

更多的 合 方式会在下章[多字段搜索](#)中介，但在此之前，我 先看 外一个重要的 特性：文本分析（text analysis）。

控制分析

只能 倒排索引表中真 存在的 ，所以保 文 在索引 与 字符串在搜索 用相同的分析 程非常重 要， 的 才能 匹配倒排索引中的 。

尽管是在 文 ，不 分析器可以由 个字段决定。 个字段都可以有不同的分析器，既可以通 配置 字段指定分析器，也可以使用更高 的 型 (type)、索引 (index) 或 点 (node) 的 配置。在索 引 ，一个字段 是根据配置或 分析器分析的。

例如 `my_index` 新 一个字段：

```
PUT /my_index/_mapping/my_type
{
  "my_type": {
    "properties": {
      "english_title": {
        "type": "string",
        "analyzer": "english"
      }
    }
  }
}
```

在我 就可以通 使用 `analyze` API 来分析 `Foxes`，而比 `english_title` 字段和 `title` 字段在索引 的分析 果：

```
GET /my_index/_analyze
{
  "field": "my_type.title", ①
  "text": "Foxes"
}

GET /my_index/_analyze
{
  "field": "my_type.english_title", ②
  "text": "Foxes"
}
```

① 字段 `title`，使用 的 `standard` 准分析器，返回 `foxes`。

② 字段 `english_title`，使用 `english` 英 分析器，返回 `fox`。

意味着，如果使用底 `term` 精 `fox`，`english_title` 字段会匹配但 `title` 字段不会。

如同 `match` 的高 知道字段映射的 系，能 个被 的字段 用正 的分析器。可以使用 `validate-query` API 看 个行：

```

GET /my_index/my_type/_validate/query?explain
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "Foxes" } },
        { "match": { "english_title": "Foxes" } }
      ]
    }
  }
}

```

返回 句的 explanation 果：

```
(title:foxes english_title:fox)
```

match 个字段使用合 的分析器，以保 它在 个 都 字段使用正 的格式。

分析器

然我 可以在字段 指定分析器，但是如果 没有指定任何的分析器，那 我 如何能 定 个字
段使用的是 个分析器 ？

分析器可以从三个 面 行定：按字段 (per-field)、按索引 (per-index) 或全局 省 (global default)。Elasticsearch 会按照以下 序依次 理，直到它 到能 使用的分析器。索引 的 序如下：

- 字段映射里定 的 **analyzer**，否
- 索引 置中名 **default** 的分析器，
- **standard** 准分析器

在搜索 ， 序有些 不同：

- 自己定 的 **analyzer**，否
- 字段映射里定 的 **analyzer**，否
- 索引 置中名 **default** 的分析器，
- **standard** 准分析器

有 ， 在索引 和搜索 使用不同的分析器是合理的。我 可能要想 同 建索引（例如，所有 **quick** 出 的地方，同 也 **fast**、**rapid** 和 **speedy** 建索引）。但在搜索 ， 我 不需要搜索所有的同 ， 取而代之的是 用 入的 是否是 **quick**、**fast**、**rapid** 或 **speedy**。

了区分，Elasticsearch 也支持一个可 的 **search_analyzer** 映射，它 会 用于搜索 （ **analyzer** 用于索引 ）。 有一个等 的 **default_search** 映射，用以指定索引 的 配置。

如果考 到 些 外参数，一个搜索 的完整 序会是下面 ：

- 自己定 的 `analyzer`，否
- 字段映射里定 的 `search_analyzer`，否
- 字段映射里定 的 `analyzer`，否
- 索引 置中名 `default_search` 的分析器，
- 索引 置中名 `default` 的分析器，
- `standard` 准分析器

分析器配置 践

就可以配置分析器地方的数量而言是十分 人的，但是 非常 。

保持

多数情况下，会提前知道文 会包括 些字段。最 的途径就是在 建索引或者 加 型映射 ， 个全文字段 置分析器。 方式尽管有点麻 ，但是它 我 可以清楚的看到 个字段 个分析器是如何置的。

通常，多数字符串字段都是 `not_analyzed` 精 字段，比如 `(tag)` 或枚 (enum)，而且更多的全文字段会使用 的 `standard` 分析器或 `english` 或其他某 言的分析器。只需要 少数一 个字段指定自定 分析：或 `title` 字段需要以支持 入即 (*find-as-you-type*) 的方式 行索引。

可以在索引 置中， 大部分的字段 置 想指定的 `default` 分析器。然后在字段 置中， 某一 个字段配置需要指定的分析器。

NOTE 于和 相 的日志数据，通常的做法是 天自行 建索引，由于 方式不是从 建 的索引， 然可以用 `{ref}/indices-templates.html`[索引模板 (Index Template)] 新建的索引指定配置和映射。

被破坏的相 度！

在 更 的 多字段搜索 之前，我 先快速解 一下 什 只在主分片上 建 索引。

用 会 不 的抱怨无法按相 度排序并提供 短的重 :用 索引了一些文 ， 行一个 的，然后 明 低相 度的 果出 在高相 度 果之上。

了理解 什 会 ，可以 想，我 在 个主分片上 建了索引和 共 10 个文 ，其中 6 个文 有 `foo`。可能是分片 1 有其中 3 个 `foo` 文 ，而分片 2 有其中 外 3 个文 ， 句 ，所有文 是均 分布存 的。

在 什 是相 度？ 中，我 描述了 Elasticsearch 使用的相似度算法， 个算法叫做 /逆向文 率 或 TF/IDF 。 是 算某个 在当前被 文 里某个字段中出 的 率，出 的 率越高，文 越相 。 逆向文 率 将 某个 在索引内所有文 出 的百分数 考 在内，出 的 率越高，它的 重就越低。

但是由于性能原因， Elasticsearch 不会 算索引内所有文 的 IDF 。相反， 个分片会根据 分片 内的所有文 算一个本地 IDF 。

因文是均匀分布存的，一个分片的 IDF 是相同的。相反，想如果有 5 个 `foo` 文存于分片 1，而第 6 个文存于分片 2，在这种情况下，`foo` 在一个分片里非常普通（所以不那么重要），但是在另一个分片里非常少见（所以会显得更重要）。这些 IDF 之间的差会导致不正确的结果。

在使用中，并不是一个，本地和全局的 IDF 的差会随着索引里文档数的增多而消失，在真实世界的数据量下，局部的 IDF 会被迅速均化，所以上述并不是相度被破坏所致的，而是由于数据太少。

为了，我可以通方式解决这个问题。第一是只在主分片上建索引，正如 `match` 里介的那样，如果只有一个分片，那本地的 IDF 就是全局的 IDF。

第二个方式就是在搜索求后添加 `?search_type=dfs_query_then_fetch`，`dfs` 是指分布式搜索 (*Distributed Frequency Search*)，它告诉 Elasticsearch，先分得各个分片本地的 IDF，然后根据结果再算整个索引的全局 IDF。

TIP 不要在生产环境上使用 `dfs_query_then_fetch`。完全没有必要。只要有足够的数据就能保证是均匀分布的。没有理由一个例外加上 DFS。

多字段搜索

很少是一句的 `match` 匹配。通常我需要用相同或不同的字符串一个或多个字段，也就是，需要多个句子以及它们相度分行合理的合并。

有时候或我正作者 Leo Tolstoy 写的一本名 *War and Peace*（争与和平）的。或我正用“minimum should match”（最少匹配）的方式在文中或面内容行搜索，或我正在搜索所有名字 John Smith 的用。

在本章，我会介绍多种搜索的工具及在特定情况下采用的解决方案。

多字符串

最简单的多字段可以将搜索映射到具体的字段。如果我知道 *War and Peace* 是，Leo Tolstoy 是作者，很容易就能把一个条件用 `match` 句表示，并将它用 `bool` 合起来：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "War and Peace" } },
        { "match": { "author": "Leo Tolstoy" } }
      ]
    }
  }
}
```

`bool` 采取 *more-matches-is-better* 匹配越多越好的方式，所以一条 `match` 句的结果会被加在一起，从而一个文提供最高的分数 `_score`。能与一条句同匹配的文比只与一条

句匹配的文 得分要高。

当然，并不是只能使用 `match` 句：可以用 `bool` 来包 合任意其他 型的 ，甚至包括其他的 `bool` 。我 可以在上面的示例中添加一条 句来指定 者版本的偏好：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "War and Peace" }},
        { "match": { "author": "Leo Tolstoy" }},
        { "bool": {
          "should": [
            { "match": { "translator": "Constance Garnett" }},
            { "match": { "translator": "Louise Maude" }}
          ]
        }}
      ]
    }
  }
}
```

什 将 者条件 句放入 一个独立的 `bool` 中 ？所有的四个 `match` 都是 `should` 句，所以 什 不将 `translator` 句与其他如 `title`、`author` 的 句放在同一 ？

答案在于 分的 算方式。 `bool` 行 个 `match` ，再把 分加在一起，然后将 果与所有匹配的 句数量相乘，最后除以所有的 句数量。 于同一 的 条 句具有相同的 重。在前面 个例子中，包含 `translator` 句的 `bool` ，只占 分的三分之一。如果将 `translator` 句与 `title` 和 `author` 条 句放入同一 ，那 `title` 和 `author` 句只 献四分之一 分。

句的 先

前例中 条 句 献三分之一 分的 方式可能并不是我 想要的，我 可能 title 和 author 条 句更感 趣， 就需要 整 ，使 `title` 和 `author` 句相 来 更重要。

在武器 中，最容易使用的就是 `boost` 参数。 了提升 `title` 和 `author` 字段的 重， 它 分配的 `boost` 大于 1 :

```

GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { ①
          "title": {
            "query": "War and Peace",
            "boost": 2
          }
        }},
        { "match": { ①
          "author": {
            "query": "Leo Tolstoy",
            "boost": 2
          }
        }},
        { "bool": { ②
          "should": [
            { "match": { "translator": "Constance Garnett" } },
            { "match": { "translator": "Louise Maude" } }
          ]
        }}
      ]
    }
  }
}

```

① title 和 author 句的 boost 2。

② 嵌套 bool 句 的 boost 1。

要 取 boost 参数 “最佳” , 的方式就是不断 : 定 boost , 行 , 如此反 。 boost 比 合理的区 于 1 到 10 之 , 当然也有可能是 15 。如果 boost 指定比 更高的 , 将不会 最 的 分 果 生更大影 , 因 分是被 一化的 (normalized) 。

字符串

bool 是多 句 的主干。它的 用 景很多, 特 是当需要将不同 字符串映射到不同字段的 候。

在于, 目前有些用 期望将所有的搜索 堆 到 个字段中, 并期望 用程序能 他 提供正 的 果。有意思的是多字段搜索的表 通常被称 高 (Advanced Search) —— 只是因 它 用 而言是高 的, 而多字段搜索的 却非常 。

于多 (multiword) 、多字段 (multifield) 来 , 不存在 的 万能 方案。 了 得最好 果, 需要 了解我 的数据 , 并了解如何使用合 的工具。

了解我 的数据

当用 入了 个字符串 的 候, 通常会遇到以下三 情形 :

最佳字段

当搜索具体概念的时候，比如“brown fox”，比各自独立的更有意。像 `title` 和 `body` 的字段，尽管它们是相同的，但同时又彼此相互竞争。文本在相同字段中包含的越多越好，部分也来自于最匹配字段。

多字段

为了相度行微，常用的一个技巧就是将相同的数据索引到不同的字段，它们各自具有独立的分析。

主字段可能包括它的源、同音以及音或口音，被用来匹配尽可能多的文本。

相同的文本被索引到其他字段，以提供更精确的匹配。一个字段可以包括未干提取的原词，一个字段包括其他源、口音，有一个字段可以提供相似性信息的瓦片（shingles）。

其他字段是作匹配个体文本提高相度分的信号，匹配字段越多越好。

混合字段

于某些主体，我需要在多个字段中指定其信息，各个字段都只能作整体的一部分：

- Person：`first_name` 和 `last_name`（人：名和姓）
- Book：`title`、`author` 和 `description`（书名：作者、描述）
- Address：`street`、`city`、`country` 和 `postcode`（地址：街道、市、国家和邮政编码）

在这些情况下，我希望在任何一些列出的字段中找到尽可能多的词，有如在一个大字段中进行搜索，一个大字段包括了所有列出的字段。

上述所有都是多字段，但每个具体字段都要求使用不同策略。本章后面的部分，我会依次介绍一个策略。

最佳字段

假设有个网站允许用搜索博客的内容，以下面两篇博客内容为例：

```
PUT /my_index/my_type/1
{
    "title": "Quick brown rabbits",
    "body": "Brown rabbits are commonly seen."
}

PUT /my_index/my_type/2
{
    "title": "Keeping pets healthy",
    "body": "My quick brown fox eats rabbits on a regular basis."
}
```

用户输入“Brown fox”然后点击搜索按钮。事先，我并不知道用哪个搜索词会在 `title` 或是在 `body` 字段中被找到，但是，用很有可能是想搜索相同的词。用肉眼判断，文档 2 的匹配度更高，因为它同时包括要找的这个词：

在 行以下 `bool` :

```
{  
  "query": {  
    "bool": {  
      "should": [  
        { "match": { "title": "Brown fox" }},  
        { "match": { "body": "Brown fox" }}  
      ]  
    }  
  }  
}
```

但是我 的 果是文 1 的 分更高：

```
{  
  "hits": [  
    {  
      "_id": "1",  
      "_score": 0.14809652,  
      "_source": {  
        "title": "Quick brown rabbits",  
        "body": "Brown rabbits are commonly seen."  
      }  
    },  
    {  
      "_id": "2",  
      "_score": 0.09256032,  
      "_source": {  
        "title": "Keeping pets healthy",  
        "body": "My quick brown fox eats rabbits on a regular basis."  
      }  
    }  
  ]  
}
```

了理解 致 的原因，需要回想一下 `bool` 是如何 算 分的：

1. 它会 行 `should` 句中的 个 。
2. 加和 个 的 分。
3. 乘以匹配 句的 数。
4. 除以所有 句 数 (里 :2) 。

文 1 的 个字段都包含 `brown` 个 ，所以 个 `match` 句都能成功匹配并且有一个 分。文 2 的 `body` 字段同 包含 `brown` 和 `fox` 个 ，但 `title` 字段没有包含任何 。 ， `body` 果中的高分，加上 `title` 中的 0 分，然后乘以二分之一，就得到比文 1 更低的整体 分。

在本例中，`title` 和 `body` 字段是相互 竞争的 系，所以就需要 到 一个最佳匹配 的字段。

如果不是 将 个字段的 分 果加在一起，而是将 最佳匹配 字段的 分作 的整体 分，果会 ？ 返回的 果可能是： 同 包含 `brown` 和 `fox` 的 个字段比反 出 相同 的多个不同字段有更高的相 度。

dis_max

不使用 `bool` ， 可以使用 `dis_max` 即分 最大化 (*Disjunction Max Query*) 。分 (Disjunction) 的意思是 或 (or) ， 与可以把 合 (conjunction) 理解成 与 (and) 相 。分最大化 (Disjunction Max Query) 指的是： 将任何与任一 匹配的文 作 果返回，但只将最佳匹配的 分作 的 分 果返回：

```
{  
  "query": {  
    "dis_max": {  
      "queries": [  
        { "match": { "title": "Brown fox" }},  
        { "match": { "body": "Brown fox" }}  
      ]  
    }  
  }  
}
```

得到我 想要的 果 :

```
{  
  "hits": [  
    {  
      "_id": "2",  
      "_score": 0.21509302,  
      "_source": {  
        "title": "Keeping pets healthy",  
        "body": "My quick brown fox eats rabbits on a regular basis."  
      }  
    },  
    {  
      "_id": "1",  
      "_score": 0.12713557,  
      "_source": {  
        "title": "Quick brown rabbits",  
        "body": "Brown rabbits are commonly seen."  
      }  
    }  
  ]  
}
```

最佳字段

当用 搜索 “quick pets” 会 生什 ？在前面的例子中， 个文 都包含 `quick`，但是只有文 2 包含 `pets`， 个文 中都不具有同 包含 个 的相同字段。

如下，一个 的 `dis_max` 会采用 个最佳匹配字段，而忽略其他的匹配：

```
{  
  "query": {  
    "dis_max": {  
      "queries": [  
        { "match": { "title": "Quick pets" }},  
        { "match": { "body": "Quick pets" }}  
      ]  
    }  
  }  
}
```

```
{  
  "hits": [  
    {  
      "_id": "1",  
      "_score": 0.12713557, ①  
      "_source": {  
        "title": "Quick brown rabbits",  
        "body": "Brown rabbits are commonly seen."  
      }  
    },  
    {  
      "_id": "2",  
      "_score": 0.12713557, ①  
      "_source": {  
        "title": "Keeping pets healthy",  
        "body": "My quick brown fox eats rabbits on a regular basis."  
      }  
    }  
  ]  
}
```

① 注意 个 分是完全相同的。

我 可能期望同 匹配 `title` 和 `body` 字段的文 比只与一个字段匹配的文 的相 度更高，但事 并非如此，因 `dis_max` 只会 地使用 个 最佳匹配 句的 分 `_score` 作 整体 分。

`tie_breaker` 参数

可以通 指定 `tie_breaker` 个参数将其他匹配 句的 分也考 其中：

```
{
  "query": {
    "dis_max": {
      "queries": [
        { "match": { "title": "Quick pets" }},
        { "match": { "body": "Quick pets" }}
      ],
      "tie_breaker": 0.3
    }
  }
}
```

果如下：

```
{
  "hits": [
    {
      "_id": "2",
      "_score": 0.14757764, ①
      "_source": {
        "title": "Keeping pets healthy",
        "body": "My quick brown fox eats rabbits on a regular basis."
      }
    },
    {
      "_id": "1",
      "_score": 0.124275915, ①
      "_source": {
        "title": "Quick brown rabbits",
        "body": "Brown rabbits are commonly seen."
      }
    }
  ]
}
```

① 文 2 的相 度比文 1 略高。

`tie_breaker` 参数提供了一 `dis_max` 和 `bool` 之 的折中 ，它的 分方式如下：

1. 得最佳匹配 句的 分 `_score`。
2. 将其他匹配 句的 分 果与 `tie_breaker` 相乘。
3. 以上 分求和并 化。

有了 `tie_breaker`，会考 所有匹配 句，但最佳匹配 句依然占最 果里的很大一部分。

NOTE

`tie_breaker` 可以是 0 到 1 之 的浮点数，其中 0 代表使用 `dis_max` 最佳匹配句的普通，1 表示所有匹配句同等重要。最佳的精 需要根据数据与得出，但是合理 与零接近（于 0.1 - 0.4 之），就不会 覆 `dis_max` 最佳匹配性 的根本。

multi_match

`multi_match` 能在多个字段上反 行相同 提供了一 便捷方式。

NOTE

`multi_match` 多匹配 的 型有多 ，其中的三 恰巧与 了解我 的数据 中介 的三个 景 ，即：`best_fields` 、 `most_fields` 和 `cross_fields` （最佳字段、多数字段、跨字段）。

情况下， 的 型是 `best_fields`， 表示它会 个字段生成一个 `match` ，然后将它 合到 `dis_max` 的内部，如下：

```
{  
  "dis_max": {  
    "queries": [  
      {  
        "match": {  
          "title": {  
            "query": "Quick brown fox",  
            "minimum_should_match": "30%"  
          }  
        }  
      },  
      {  
        "match": {  
          "body": {  
            "query": "Quick brown fox",  
            "minimum_should_match": "30%"  
          }  
        }  
      },  
    ],  
    "tie_breaker": 0.3  
  }  
}
```

上面 个 用 `multi_match` 重写成更 的形式：

```
{
  "multi_match": {
    "query": "Quick brown fox",
    "type": "best_fields", ①
    "fields": [ "title", "body" ],
    "tie_breaker": 0.3,
    "minimum_should_match": "30%" ②
  }
}
```

① `best_fields` 型是 ，可以不指定。

② 如 `minimum_should_match` 或 `operator` 的参数会被 到生成的 `match` 中。

字段名称的模糊匹配

字段名称可以用模糊匹配的方式 出：任何与模糊模式正 匹配的字段都会被包括在搜索条件中，例如可以使用以下方式同 匹配 `book_title` 、 `chapter_title` 和 `section_title` （ 名、章名、 名）三个字段：

```
{
  "multi_match": {
    "query": "Quick brown fox",
    "fields": "*_title"
  }
}
```

提升 个字段的 重

可以使用 ^ 字符 法 个字段提升 重，在字段名称的末尾添加 `^boost`，其中 `boost` 是一个浮点数：

```
{
  "multi_match": {
    "query": "Quick brown fox",
    "fields": [ "*_title", "chapter_title^2" ] ①
  }
}
```

① `chapter_title` 个字段的 `boost` 2，而其他 个字段 `book_title` 和 `section_title` 字段的 `boost` 1。

多字段

全文搜索被称作是 召回率 (*Recall*) 与 精 率 (*Precision*) 的 ；召回率 ——返回所有的相 文 ；精 率 ——不返回无 文 。目的是在 果的第一 中 用 呈 最 相 的文 。

了提高召回率的效果，我 大搜索 ——不 返回与用 搜索 精 匹配的文 ， 会返回我 与 相 的所有文 。如果一个用 搜索 “quick brown box”，一个包含 fast foxes 的文 被

是非常合理的返回 果。

如果包含 **fast foxes** 的文 是能 到的唯一相 文 ，那 它会出 在 果列表的最上面，但是，如果有 100 个文 都出 了 **quick brown fox**，那 个包含 **fast foxes** 的文 当然会被 是次相 的，它可能 于返回 果列表更下面的某个地方。当包含了很多潜在匹配之后，我 需要将最匹配的几个置于 果列表的 部。

提高全文相 性精度的常用方式是 同一文本建立多 方式的索引， 方式都提供了一个不同的相 度信 号 *signal*。主字段会以尽可能多的形式去匹配尽可能多的文 。 个例子，我 可以 行以下操作：

- 使用 干提取来索引 **jumps**、**jumping** 和 **jumped** 的 ，将 **jump** 作 它 的 根形式。 即使用 搜索 **jumped**，也 是能 到包含 **jumping** 的匹配的文 。
- 将同 包括其中，如 **jump**、**leap** 和 **hop** 。
- 移除 音或口音：如 **ésta**、**está** 和 **esta** 都会以无 音形式 **esta** 来索引。

尽管如此，如果我 有 个文 ，其中一个包含 **jumped**，一个包含 **jumping**，用 很可能期望前者能排的更高，因 它正好与 入的搜索条件一致。

了 到目的，我 可以将相同的文本索引到其他字段从而提供更 精 的匹配。一个字段可能是 干未 提取 的版本， 一个字段可能是 音 的原始 ，第三个可能使用 **shingles** 提供 相似性 信息。 些附加的字段可以看成提高 个文 的相 度 分的信号 *signals*，能匹配字段的越多越好。

一个文 如果与广度匹配的主字段相匹配，那 它会出 在 果列表中。如果文 同 又与 *signal* 信号字段匹配，那 它会 得 外加分，系 会提升它在 果列表中的位置。

我 会在本 后 同 、 相似性、部分匹配以及其他潜在的信号 行 ，但 里只使用 干已提取 (stemmed) 和未提取 (unstemmed) 的字段作 例子来 明 技 。

多字段映射

首先要做的事情就是 我 的字段索引 次：一次使用 干模式以及一次非 干模式。 了做到 点，采用 **multifields** 来 ，已 在 **multifields** 有所介 ：

```

DELETE /my_index

PUT /my_index
{
  "settings": { "number_of_shards": 1 }, ①
  "mappings": {
    "my_type": {
      "properties": {
        "title": { ②
          "type": "string",
          "analyzer": "english",
          "fields": {
            "std": { ③
              "type": "string",
              "analyzer": "standard"
            }
          }
        }
      }
    }
  }
}

```

① 参考 被破坏的相 度。

② title 字段使用 english 英 分析器来提取 干。

③ title.std 字段使用 standard 准分析器，所以没有 干提取。

接着索引一些文 :

```

PUT /my_index/my_type/1
{ "title": "My rabbit jumps" }

PUT /my_index/my_type/2
{ "title": "Jumping jack rabbits" }

```

里用一个 match title 字段是否包含 jumping rabbits (跳 的兔子) :

```

GET /my_index/_search
{
  "query": {
    "match": {
      "title": "jumping rabbits"
    }
  }
}

```

因 有了 english 分析器， 个 是在 以 jump 和 rabbit 个被提取 的文 。 个文 的 title

字段都同 包括 个 , 所以 个文 得到的 分也相同 :

```
{  
  "hits": [  
    {  
      "_id": "1",  
      "_score": 0.42039964,  
      "_source": {  
        "title": "My rabbit jumps"  
      }  
    },  
    {  
      "_id": "2",  
      "_score": 0.42039964,  
      "_source": {  
        "title": "Jumping jack rabbits"  
      }  
    }  
  ]  
}
```

如果只是 `title.std` 字段, 那 只有文 2 是匹配的。尽管如此, 如果同 个字段, 然后使用 `bool` 将 分果合并, 那 个文 都是匹配的 (`title` 字段的作用), 而且文 2 的相 度分更高 (`title.std` 字段的作用) :

```
GET /my_index/_search  
{  
  "query": {  
    "multi_match": {  
      "query": "jumping rabbits",  
      "type": "most_fields", ①  
      "fields": [ "title", "title.std" ]  
    }  
  }  
}
```

① 我 希望将所有匹配字段的 分合并起来, 所以使用 `most_fields` 型。`multi_match` 用 `bool` 将 个字段 句包在里面, 而不是使用 `dis_max` 。

```
{
  "hits": [
    {
      "_id": "2",
      "_score": 0.8226396, ①
      "_source": {
        "title": "Jumping jack rabbits"
      }
    },
    {
      "_id": "1",
      "_score": 0.10741998, ①
      "_source": {
        "title": "My rabbit jumps"
      }
    }
  ]
}
```

① 文 2 在的 分要比文 1 高。

用广度匹配字段 `title` 包括尽可能多的文——以提升召回率——同 又使用字段 `title.std` 作 信号 将相 度更高的文 置于 果 部。

个字段 于最 分的 献可以通 自定 `boost` 来控制。比如，使 `title` 字段更 重要， 同 也降低了其他信号字段的作用：

```
GET /my_index/_search
{
  "query": {
    "multi_match": {
      "query": "jumping rabbits",
      "type": "most_fields",
      "fields": [ "title^10", "title.std" ] ①
    }
  }
}
```

① `title` 字段的 `boost` 的 10 使它比 `title.std` 更重要。

跨字段 体搜索

在 一 普遍的搜索模式：跨字段 体搜索（cross-fields entity search）。在如 `person`、`product` 或 `address`（人、 品或地址） 的 体中，需要使用多个字段来唯一 它的信息。`person` 体可能是 索引的：

```
{  
  "firstname": "Peter",  
  "lastname": "Smith"  
}
```

或地址：

```
{  
  "street": "5 Poland Street",  
  "city": "London",  
  "country": "United Kingdom",  
  "postcode": "W1V 3DG"  
}
```

与之前描述的 多字符串 很像，但 存在着巨大的区 。在 多字符串 中，我 个字段使用不同的字符串，在本例中，我 想使用 个字符串在多个字段中 行搜索。

我 的用 可能想搜索 “Peter Smith” 个人，或 “Poland Street W1V” 个地址， 些 出 在不同的字段中，所以如果使用 `dis_max` 或 `best_fields` 去 一个 最佳匹配字段 然是个 的方式。

的方式

依次 个字段并将 个字段的匹配 分 果相加，听起来真像是 `bool` :

```
{  
  "query": {  
    "bool": {  
      "should": [  
        { "match": { "street": "Poland Street W1V" }},  
        { "match": { "city": "Poland Street W1V" }},  
        { "match": { "country": "Poland Street W1V" }},  
        { "match": { "postcode": "Poland Street W1V" }}  
      ]  
    }  
  }  
}
```

个字段重 字符串会使 瞬 得冗 ，可以采用 `multi_match` ，将 `type` 置成 `most_fields` 然后告 Elasticsearch 合并所有匹配字段的 分：

```
{
  "query": {
    "multi_match": {
      "query": "Poland Street W1V",
      "type": "most_fields",
      "fields": [ "street", "city", "country", "postcode" ]
    }
  }
}
```

most_fields 方式的

用 most_fields 方式搜索也存在某些问题，一些问题并不会马上：

- 它是多数字段匹配任意一个的，而不是在所有字段中找到最匹配的。
- 它不能使用 operator 或 minimum_should_match 参数来降低次相关结果造成的尾效应。
- 由于一个字段是唯一的，而且它之间的相互影响会导致不好的排序结果。

字段中心式

以上三个源于 most_fields 的问题都因它是字段中心式 (field-centric) 而不是术语中心式 (term-centric) 的：当真正感兴趣的匹配的时候，它只关心的是最匹配的字段。

NOTE best_fields 型也是字段中心式的，它也存在类似的问题。

首先看这些问题存在的原因，再想如何解决它们。

1：在多个字段中匹配相同的值

回想一下 most_fields 是如何工作的：Elasticsearch 为每个字段生成独立的 match，再用 bool 将它们包起来。

可以通过 validate-query API 看：

```
GET /_validate/query?explain
{
  "query": {
    "multi_match": {
      "query": "Poland Street W1V",
      "type": "most_fields",
      "fields": [ "street", "city", "country", "postcode" ]
    }
  }
}
```

生成 explanation 解释：

```
(street:poland street:street street:w1v)
(city:poland city:street city:w1v)
(country:poland country:street country:w1v)
(postcode:poland postcode:street postcode:w1v)
```

可以，一个字段都与 `poland` 匹配的文 要比一个字段同 匹配 `poland` 与 `street` 文 的 分高。

2 : 剪掉 尾

在 `匹配精度` 中，我 使用 `and` 操作符或 置 `minimum_should_match` 参数来消除 果中几乎不相的 尾，或 可以 以下方式：

```
{
  "query": {
    "multi_match": {
      "query": "Poland Street W1V",
      "type": "most_fields",
      "operator": "and", ①
      "fields": [ "street", "city", "country", "postcode" ]
    }
  }
}
```

① 所有 必 呈 。

但是 于 `best_fields` 或 `most_fields` 些参数会在 `match` 生成 被 入， 个 的 `explanation` 解 如下：

```
(+street:poland +street:street +street:w1v)
(+city:poland +city:street +city:w1v)
(+country:poland +country:street +country:w1v)
(+postcode:poland +postcode:street +postcode:w1v)
```

句 ， 使用 `and` 操作符要求所有 都必 存在于 相同字段 ， 然是不 的！可能就不存在能与 个 匹配的文 。

3 :

在 `什 是相` 中，我 解 个 使用 TF/IDF 相似度算法 算相 度 分：

一个 在 个文 的某个字段中出 的 率越高， 个文 的相 度就越高。

逆向文 率

一个 在所有文 某个字段索引中出 的 率越高， 个 的相 度就越低。

当搜索多个字段 ， TF/IDF 会 来某些令人意外的 果。

想想用字段 `first_name` 和 `last_name` “Peter Smith”的例子，Peter 是个平常的名 Smith 也是平常的姓，两者都具有低的 IDF。但当索引中有外一个人的名字是“Smith Williams”，Smith 作名来很不平常，以致它有一个高的 IDF！

下面一个的可能会在果中将“Smith Williams”置于“Peter Smith”之上，尽管事实上是第二个人比第一个人更匹配。

```
{  
  "query": {  
    "multi_match": {  
      "query": "Peter Smith",  
      "type": "most_fields",  
      "fields": ["*_name"]  
    }  
  }  
}
```

里的 `smith` 在名字段中具有高 IDF，它会削弱“Peter”作名和“Smith”作姓低 IDF 的所起作用。

解决方案

存在一些是因我在理着多个字段，如果将所有这些字段合成一个字段，就会消失。可以 person 文添加 `full_name` 字段来解决这个问题：

```
{  
  "first_name": "Peter",  
  "last_name": "Smith",  
  "full_name": "Peter Smith"  
}
```

当 `full_name` 字段：

- 具有更多匹配的文会比只有一个重匹配的文更重要。
- `minimum_should_match` 和 `operator` 参数会像期望那样工作。
- 姓和名的逆向文率被合并，所以 Smith 到底是作姓是作名出，都会得无必要。

做当然是可行的，但我并不太喜欢存冗余数据。取而代之的是 Elasticsearch 可以提供一个解决方案——一个在索引，而一个是在搜索——随后会把它。

自定 `_all` 字段

在 `all-field` 字段中，我解`_all`字段的索引方式是将所有其他字段的作一个大字符串索引的。然而做并不十分活，为了活我可以人名添加一个自定`_all`字段，再地址添加一个`_all`字段。

Elasticsearch 在字段映射中我提供 `copy_to` 参数来这个功能：

```
PUT /my_index
{
  "mappings": {
    "person": {
      "properties": {
        "first_name": {
          "type": "string",
          "copy_to": "full_name" ①
        },
        "last_name": {
          "type": "string",
          "copy_to": "full_name" ①
        },
        "full_name": {
          "type": "string"
        }
      }
    }
  }
}
```

① `first_name` 和 `last_name` 字段中的 值会被 复制到 `full_name` 字段。

有了 3 个映射，我 可以用 `first_name` 来 姓，用 `last_name` 来 姓，或者直接使用 `full_name` 整个姓名。

`first_name` 和 `last_name` 的映射并不影响 `full_name` 如何被索引，`full_name` 将 3 个字段的内容复制到本地，然后根据 `full_name` 的映射自行索引。

`copy_to` 置 `multi-field`无效。如果配置映射，Elasticsearch 会常。

什？多字段只是以不同方式索引“主”字段；它没有自己的数据源。也就是没有可供 `copy_to` 到一字段的数据源。

只要“主”字段 `copy_to` 就能而易的到相同的效果：

```
PUT /my_index
{
  "mappings": {
    "person": {
      "properties": {
        "first_name": {
          "type": "string",
          "copy_to": "full_name", ①
        },
        "fields": {
          "raw": {
            "type": "string",
            "index": "not_analyzed"
          }
        }
      },
      "full_name": {
        "type": "string"
      }
    }
  }
}
```

WARNING

① `copy_to` 是“主”字段，而不是多字段的

cross-fields 跨字段

自定 `all` 的方式是一个好的解决方案，只需在索引文前其置好映射。不，在搜索提供了相的解决方案：使用 `cross_fields` 型行 `multi_match`。`cross_fields` 使用中心式 (term-centric) 的方式，与 `best_fields` 和 `most_fields` 使用字段中心式 (field-centric) 的方式非常不同，它将所有字段当成一个大字段，并在一个字段中一个。

了明字段中心式 (field-centric) 与中心式 (term-centric) 方式的不同，先看看以下字段中心式的 `most_fields` 的 explanation 解：

```

GET /_validate/query?explain
{
  "query": {
    "multi_match": {
      "query": "peter smith",
      "type": "most_fields",
      "operator": "and", ①
      "fields": [ "first_name", "last_name" ]
    }
  }
}

```

① 所有 都是必 的。

于匹配的文 , `peter` 和 `smith`都必 同 出 在相同字段中, 要 是 `first_name` 字段, 要 `last_name` 字段 :

```

(+first_name:peter +first_name:smith)
(+last_name:peter +last_name:smith)

```

中心式会使用以下 :

```

+(first_name:peter last_name:peter)
+(first_name:smith last_name:smith)

```

句 , `peter` 和 `smith`都必 出 , 但是可以出 在任意字段中。

`cross_fields` 型首先分析 字符串并生成一个 列表, 然后它从所有字段中依次搜索 个 。不同的搜索方式很自然的解决了 [字段中心式](#) 三个 中的二个。剩下的 是逆向文率不同。

幸 的是 `cross_fields` 型也能解决 个 , 通 `validate-query` 可以看到 :

```

GET /_validate/query?explain
{
  "query": {
    "multi_match": {
      "query": "peter smith",
      "type": "cross_fields", ①
      "operator": "and",
      "fields": [ "first_name", "last_name" ]
    }
  }
}

```

① 用 `cross_fields` 中心式匹配。

它通过混合不同字段逆向索引文率的方式解决了 的：

```
+blended("peter", fields: [first_name, last_name])
+blended("smith", fields: [first_name, last_name])
```

句，它会同时在 `first_name` 和 `last_name` 个字段中 `smith` 的 IDF，然后用者的最小作一个字段的 IDF。果上就是 `smith` 会被既是个平常的姓，也是平常的名。

了 `cross_fields` 以最 方式工作，所有的字段都使用相同的分析器，具有相同分析器的字段会被分在一起作混合字段使用。

如果包括了不同分析的字段，它会以 `best_fields` 的相同方式被加入到果中。例如：我将 `title` 字段加到之前的 中（假 它使用的是不同的分析器），`explanation` 的解 果如下：

NOTE

```
(+title:peter +title:smith)
(
  +blended("peter", fields: [first_name, last_name])
  +blended("smith", fields: [first_name, last_name])
)
```

当在使用 `minimum_should_match` 和 `operator` 参数，点尤 重要。

按字段提高重

采用 `cross_fields` 与 自定 `_all` 字段 相比，其中一个 就是它可以在搜索 个字段提升重。

像 `first_name` 和 `last_name` 具有相同的字段并不是必 的，但如果要用 `title` 和 `description` 字段搜索 ，可能希望 `title` 分配更多的 重，同 可以使用前面介 的 ^ 符号 法来：

```
GET /books/_search
{
  "query": {
    "multi_match": {
      "query": "peter smith",
      "type": "cross_fields",
      "fields": [ "title^2", "description" ] ①
    }
  }
}
```

① `title` 字段的 重提升 2，`description` 字段的 重提升 1。

自定 字段 是否能 于多字段 ，取决于在多字段 与 字段自定 `_all` 之 代 的衡，即 解决方案会 来更大的性能 化就 一。

Exact-Value 精 字段

在 束多字段 个 之前，我 最后要 的是精 `not_analyzed` 未分析字段。将 `not_analyzed` 字段与 `multi_match` 中 `analyzed` 字段混在一起没有多大用。

原因可以通 看 的 explanation 解 得到， 想将 `title` 字段 置成 `not_analyzed`：

```
GET /_validate/query?explain
{
  "query": {
    "multi_match": {
      "query": "peter smith",
      "type": "cross_fields",
      "fields": [ "title", "first_name", "last_name" ]
    }
  }
}
```

因 `title` 字段是未分析 的，Elasticsearch 会将“peter smith” 个完整的字符串作 条件来搜索！

```
title:peter smith
(
  blended("peter", fields: [first_name, last_name])
  blended("smith", fields: [first_name, last_name])
)
```

然 个 不在 `title` 的倒排索引中，所以需要在 `multi_match` 中避免使用 `not_analyzed` 字段。

近似匹配

使用 TF/IDF 的 准全文 索将文 或者文 中的字段作一大袋的 理。`match` 可以告知我 大袋子中是否包含 的 条，但却无法告知 之 的 系。

思考下面 几个句子的不同：

- Sue ate the alligator.
- The alligator ate Sue.
- Sue never goes anywhere without her alligator-skin purse.

用 `match` 搜索 `sue alligator` 上面的三个文 都会得到匹配，但它却不能 定 个 是否只来自于一 境，甚至都不能 定是否来自于同一个段落。

理解分 之 的 系是一个 的 ，我 也无法通 一 方式去解决。但我 至少可以通 出 在彼此附近或者 是彼此相 的分 来判断一些似乎相 的分 。

个文 可能都比我 上面 个例子要 : `Sue` 和 `alligator` 个 可能会分散在其他的段落文字中，我 可能会希望得到尽可能包含 个 的文 ，但我 也同 需要 些

文 与分 有很高的相 度。

就是短 匹配或者近似匹配的所属 域。

TIP 在 一 章 , 我 是 使用 在 `match` 中 使用 的 文 作 例 子。

短 匹 配

就 像 `match` 于 全 文 索 是 一 最 常 用 的 一 , 当 想 到 彼 此 近 搜 索 的 方 法 , 就 会 想 到 `match_phrase` 。

```
GET /my_index/my_type/_search
{
  "query": {
    "match_phrase": {
      "title": "quick brown fox"
    }
  }
}
```

似 `match` , `match_phrase` 首先将 字符串 解析成一个 列 表 , 然后 些 行 搜 索 , 但 只 保 留 那 些 包 含 全 部 搜 索 , 且 位 置 与 搜 索 相 同 的 文 。 比 如 于 `quick fox` 的 短 搜 索 可 能 不 会 匹 配 到 任 何 文 , 因 没 有 文 包 含 的 `quick` 之 后 跟 着 `fox` 。

`match_phrase` 同 可 写 成 一 型 `phrase` 的 `match` :

TIP

```
"match": {
  "title": {
    "query": "quick brown fox",
    "type": "phrase"
  }
}
```

的 位 置

当 一 个 字 符 串 被 分 后 , 一 个 分 析 器 不 但 会 返回 一 个 列 表 , 而 且 会 返回 各 在 原 始 字 符 串 中 的 位 置 或 者 序 系 :

```
GET /_analyze?analyzer=standard
Quick brown fox
```

返 回 信 息 如 下 :

```
{
  "tokens": [
    {
      "token": "quick",
      "start_offset": 0,
      "end_offset": 5,
      "type": "<ALPHANUM>",
      "position": 1 ①
    },
    {
      "token": "brown",
      "start_offset": 6,
      "end_offset": 11,
      "type": "<ALPHANUM>",
      "position": 2 ①
    },
    {
      "token": "fox",
      "start_offset": 12,
      "end_offset": 15,
      "type": "<ALPHANUM>",
      "position": 3 ①
    }
  ]
}
```

① position 代表各 在原始字符串中的位置。

位置信息可以被存 在倒排索引中，因此 `match_phrase` 位置敏感的，
就可以利用位置信息去匹配包含所有 ，且各 序也与我 搜索指定一致的文 中 不 其他 。

什 是短

一个被 定 和短 `quick brown fox` 匹配的文 ，必 足以下 些要求：

- `quick`、`brown` 和 `fox` 需要全部出 在域中。
- `brown` 的位置 比 `quick` 的位置大 1。
- `fox` 的位置 比 `quick` 的位置大 2。

如果以上任何一个 不成立， 文 不能 定 匹配。

本上来，`match_phrase`是利用一低的span族(query family)去做位置敏感的匹配。Span是一的，所以它没有分段；它只指定的行精搜索。

TIP

得幸的是，`match_phrase`已足秀，大多数人是不会直接使用span。然而，在一些域，例如利索，是会采用低去行非常具体而又精心造的位置搜索。

混合起来

精短匹配或是于格了。也我想要包含`quick brown fox''`的文也能匹配`quick fox,"`，尽管情形不完全相同。

我能通过使用`slop`参数将活度引入短匹配中：

```
GET /my_index/my_type/_search
{
  "query": {
    "match_phrase": {
      "title": {
        "query": "quick fox",
        "slop": 1
      }
    }
  }
}
```

`slop`参数告诉`match_phrase`条相隔多条能将文匹配。相隔多的意思是除了文匹配需要移条多少次？

我以一个的例子始。为了`quick fox`能匹配一个包含`quick brown fox`的文，我需要`slop`的1:

	Pos 1	Pos 2	Pos 3
<hr/>			
Doc:	quick	brown	fox
<hr/>			
Query:	quick	fox	
Slop 1:	quick		fox

尽管在使用了`slop`短匹配中所有的都需要出，但是些也不必了匹配而按相同的序列排列。有了足大的`slop`，就能按照任意序排列了。

了使`fox quick`匹配我的文，我需要`slop`的3:

	Pos 1	Pos 2	Pos 3
Doc:	quick	brown	fox
Query:	fox	quick	
Slop 1:	fox quick	①	
Slop 2:	quick	fox	
Slop 3:	quick		fox

① 注意 `fox` 和 `quick` 在 中占据同 的位置。因此将 `fox quick` 序成 `quick fox` 需要 ，或者 2 的 `slop`。

多 字段

多 字段使用短 匹配 会 生奇怪的事。想象一下 索引 个文：

```
PUT /my_index/groups/1
{
  "names": [ "John Abraham", "Lincoln Smith"]
}
```

然后 行一个 `Abraham Lincoln` 的短：

```
GET /my_index/groups/_search
{
  "query": {
    "match_phrase": {
      "names": "Abraham Lincoln"
    }
  }
}
```

令人 的是， 即使 `Abraham` 和 `Lincoln` 在 `names` 数 里属于 个不同的人名， 我 的文 也匹配了 。 一切的原因在Elasticsearch数 的索引方式。

在分析 `John Abraham` 的 候， 生了如下信息：

- Position 1: `john`
- Position 2: `abraham`

然后在分析 `Lincoln Smith` 的 候， 生了：

- Position 3: `lincoln`
- Position 4: `smith`

句 ， Elasticsearch 以上数 分析生成了与分析 个字符串 `John Abraham Lincoln Smith` —

几乎完全相同的元。我的示例相的 lincoln 和 abraham，而且一个条存在，并且它正好相，所以一个匹配了。

幸的是，在的情况下有一叫做 position_increment_gap 的的解决方案，它在字段映射中配置。

```
DELETE /my_index/groups/ ①
```

```
PUT /my_index/_mapping/groups ②
```

```
{  
    "properties": {  
        "names": {  
            "type": "string",  
            "position_increment_gap": 100  
        }  
    }  
}
```

①首先除映射 groups 以及一个型内的所有文。

②然后建一个有正的新的映射 groups。

position_increment_gap 置告 Elasticsearch 数中个新元素加当前条 position 的指定。所以在我再索引 names 数，会生如下的果：

- Position 1: john
- Position 2: abraham
- Position 103: lincoln
- Position 104: smith

在我短可能无法匹配文因 abraham 和 lincoln 之距 100。为了匹配个文必添加 100 的 slop。

越近越好

于一个短排除了不包含切短的文，而近一个 `slop` 大于 0 的短将条的近度考到最相度 `_score` 中。通常置一个像 50 或者 100 的高 `slop`，能排除距太的文，但是也予了那些近的文更高的分数。

下列 quick dog 的近匹配了同包含 quick 和 dog 的文，但是也与 quick 和 dog 更加近的文更高的分数：

```

POST /my_index/my_type/_search
{
  "query": {
    "match_phrase": {
      "title": {
        "query": "quick dog",
        "slop": 50 ①
      }
    }
  }
}

```

① 注意高 slop。

```

{
  "hits": [
    {
      "_id": "3",
      "_score": 0.75, ①
      "_source": {
        "title": "The quick brown fox jumps over the quick dog"
      }
    },
    {
      "_id": "2",
      "_score": 0.28347334, ②
      "_source": {
        "title": "The quick brown fox jumps over the lazy dog"
      }
    }
  ]
}

```

① 分数 高因 `quick` 和 `dog` 很接近

② 分数 低因 `quick` 和 `dog` 分

使用 近度提高相 度

然 近 很有用，但是所有 条都出 在文 的要求 于 格了。我 全文搜索 一章的 控制精度 也是同 的：如果七个 条中有六个匹配，那 个文 用 而言就已 足 相 了，但是 `match_phrase` 可能会将它排除在外。

相比将使用 近匹配作 要求，我 可以将它作 信号— 使用， 作 多潜在 中的一个，会 个文 的最 分 做出 献(参考 多数字段)。

上我 想将多个 的分数累 起来意味着我 用 `bool` 将它 合并。

我 可以将一个 的 `match` 作 一个 `must` 子句。 个 将决定 些文 需要被包含到 果集中。

我可以用 `minimum_should_match` 参数去除尾。然后我可以以 `should` 子句的形式添加更多特定。一个匹配成功的都会加匹配文的相度。

```
GET /my_index/my_type/_search
{
  "query": {
    "bool": {
      "must": {
        "match": { ①
          "title": {
            "query": "quick brown fox",
            "minimum_should_match": "30%"
          }
        }
      },
      "should": {
        "match_phrase": { ②
          "title": {
            "query": "quick brown fox",
            "slop": 50
          }
        }
      }
    }
  }
}
```

① `must` 子句从果集中包含或者排除文本。

② `should` 子句加入了匹配到文本的相度分。

我当然可以在 `should` 子句里面添加其它的，其中一个只某一特定方面的相度。

性能化

短和近都比的 `query` 代价更高。一个 `match` 是看条是否存在倒排索引中，而一个 `match_phrase` 是必算并比多个可能重的位置。

[Lucene nightly benchmarks](#) 表明一个的 `term` 比一个短大快 10 倍，比近（有 `slop` 的短）大快 20 倍。当然，个代指的是在搜索而不是索引。

通常，短的外成本并不像些数字所暗示的那人。事实上，性能上的差距只是明一个的 `term` 有多快。准全文数据的短通常在几秒内完成，因此上都是完全可用，即使是在一个繁忙的集群上。

TIP

在某些特定病理案例下，短可能成本太高了，但比少。一个典型例子就是DNA序列，在序列里很多同的在很多位置重出。在里使用高 `slop` 会到致位置算大量加。

那我如何限制短和近的性能消耗？一有用的方法是少需要通短

的文 数。

果集重新 分

在先前的章 中 ，我 了而使用 近 来 整相 度，而不是使用它将文 从 果列表中添加或者排除。一个 可能会匹配成千上万的 果，但我 的用 很可能只 果的前几 感 趣。

一个 的 `match` 已 通 排序把包含所有含有搜索 条的文 放在 果列表的前面了。事 上，我 只想 些 部文 重新排序，来 同 匹配了短 的文 一个 外的相 度升 。

`search` API 通 重新 分 明 支持 功能。重新 分 段支持一个代 更高的 分算法—比如 `phrase` 一只是 了从 个分片中 得前 `K` 个 果。然后会根据它 的最新 分 重新排序。

求如下所示：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": { ①
      "title": {
        "query": "quick brown fox",
        "minimum_should_match": "30%"
      }
    },
    "rescore": {
      "window_size": 50, ②
      "query": { ③
        "rescore_query": {
          "match_phrase": {
            "title": {
              "query": "quick brown fox",
              "slop": 50
            }
          }
        }
      }
    }
  }
}
```

① `match` 决定 些文 将包含在最 果集中，并通 TF/IDF 排序。

② `window_size` 是 一分片 行重新 分的 部文 数量。

③ 目前唯一支持的重新打分算法就是 一个 ，但是以后会有 加更多的算法。

相

短 和 近 都很好用，但 有一个 点。它 于 格了： 了匹配短 ，所有 都必 存

在，即使使用了 `slop`。

用 `slop` 得到的 序的 活性也需要付出代，因 失去了 之 的 系。即使可以 `sue`、`alligator` 和 `ate` 相 出 的文，但无法分 是 `Sue ate` 是 `alligator ate`。

当 相互 合使用的 候，表 的含 比 独使用更 富。 个子句 `I'm not happy I'm working` 和 `I'm happy I'm not working` 包含相同的 ，也 有相同的 近度，但含 截然不同。

如果索引 而不是索引独立的 ，就能 些 的上下文尽可能多的保留。

句子 `Sue ate the alligator`，不 要将 一个 （或者 *unigram*）作 索引

```
["sue", "ate", "the", "alligator"]
```

也要将 个 以及它的 近 作 个 索引：

```
["sue ate", "ate the", "the alligator"]
```

些 （或者 *bigrams*）被称 *shingles*。

Shingles 不限于 ； 也可以索引三个 （*trigrams*）：

TIP

```
["sue ate the", "ate the alligator"]
```

Trigrams 提供了更高的精度，但是也大大 加了索引中唯一的数量。在大多数情况下，Bigrams 就 了。

当然，只有当用 入的 内容和在原始文 中 序相同，shingles 才是有用的； `sue alligator` 的 可能会匹配到 个 ，但是不会匹配任何 shingles。

幸 的是，用 向于使用和搜索数据相似的 造来表 搜索意 。但 一点很重要：只是索引 bigrams 是不 的；我 然需要 unigrams，但可以将匹配 bigrams 作 加相 度 分的信号。

生成 Shingles

Shingles 需要在索引 作 分析 程的一部分被 建。我 可以将 unigrams 和 bigrams 都索引到 个字段中， 但将它 分 保存在能被独立 的字段会更清晰。unigrams 字段将 成我 搜索的基 部分，而 bigrams 字段用来提高相 度。

首先，我 需要在 建分析器 使用 `shingle` 元 器：

```

DELETE /my_index

PUT /my_index
{
  "settings": {
    "number_of_shards": 1, ①
    "analysis": {
      "filter": {
        "my_shingle_filter": {
          "type": "shingle",
          "min_shingle_size": 2, ②
          "max_shingle_size": 2, ②
          "output_unigrams": false ③
        }
      },
      "analyzer": {
        "my_shingle_analyzer": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "my_shingle_filter" ④
          ]
        }
      }
    }
  }
}

```

① 参考 被破坏的相 度！。

② 最小/最大的 shingle 大小是 2，所以 上不需要 置。

③ shingle 元 器 出 unigrams，但是我 想 unigrams 和 bigrams 分 。

④ my_shingle_analyzer 使用我 常 的 my_shingles_filter 元 器。

首先，用 analyze API 下分析器：

```

GET /my_index/_analyze?analyzer=my_shingle_analyzer
Sue ate the alligator

```

果然，我 得到了 3 个：

- sue ate
- ate the
- the alligator

在我 可以 建一个使用新的分析器的字段。

多字段

我曾到将 unigrams 和 bigrams 分索引更清晰，所以 `title` 字段将建成一个多字段（参考 [\[multi-fields\]](#)）：

```
PUT /my_index/_mapping/my_type
{
  "my_type": {
    "properties": {
      "title": {
        "type": "string",
        "fields": {
          "shingles": {
            "type": "string",
            "analyzer": "my_shingle_analyzer"
          }
        }
      }
    }
  }
}
```

通过一个映射，JSON 文中的 `title` 字段将会被以 unigrams (`title`) 和 bigrams (`title.shingles`) 被索引，意味着可以独立地一些字段。

最后，我可以索引以下示例文：

```
POST /my_index/my_type/_bulk
{ "index": { "_id": 1 } }
{ "title": "Sue ate the alligator" }
{ "index": { "_id": 2 } }
{ "title": "The alligator ate Sue" }
{ "index": { "_id": 3 } }
{ "title": "Sue never goes anywhere without her alligator skin purse" }
```

搜索 Shingles

为了理解添加 `shingles` 字段的好处，我首先来看 `The hungry alligator ate Sue` 行的结果：

```

GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": "the hungry alligator ate sue"
    }
  }
}

```

↑ 返回了所有的三个文档，但是注意文档 1 和 2 有相同的相 度 分因为它们包含了相同的：

```

{
  "hits": [
    {
      "_id": "1",
      "_score": 0.44273707, ①
      "_source": {
        "title": "Sue ate the alligator"
      }
    },
    {
      "_id": "2",
      "_score": 0.44273707, ①
      "_source": {
        "title": "The alligator ate Sue"
      }
    },
    {
      "_id": "3", ②
      "_score": 0.046571054,
      "_source": {
        "title": "Sue never goes anywhere without her alligator skin purse"
      }
    }
  ]
}

```

① 三个文档都包含 `the`、`alligator` 和 `ate`，所以 得相同的 分。

② 我 可以通过 置 `minimum_should_match` 参数排除文档 3，参考 [控制精度](#)。

在 在 里添加 `shingles` 字段。不要忘了在 `shingles` 字段上的匹配是充当一个信号— 为了提高相 度 分—所以我 然需要将基本 `title` 字段包含到 中：

```

GET /my_index/my_type/_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "title": "the hungry alligator ate sue"
        }
      },
      "should": {
        "match": {
          "title.shingles": "the hungry alligator ate sue"
        }
      }
    }
  }
}

```

然匹配到了所有的 3 个文 , 但是文 2 在排到了第一名因 它匹配了 shingled ate sue.

```

{
  "hits": [
    {
      "_id": "2",
      "_score": 0.4883322,
      "_source": {
        "title": "The alligator ate Sue"
      }
    },
    {
      "_id": "1",
      "_score": 0.13422975,
      "_source": {
        "title": "Sue ate the alligator"
      }
    },
    {
      "_id": "3",
      "_score": 0.014119488,
      "_source": {
        "title": "Sue never goes anywhere without her alligator skin purse"
      }
    }
  ]
}

```

即使 包含的 hungry 没有在任何文 中出 , 我 然使用 近度返回了最相 的文 。

Performance性能

shingles 不 比短 更 活，而且性能也更好。 shingles 跟一个 的 `match` 一高效，而不用 次搜索花 短 的代 。只是在索引期 因 更多 需要被索引会付出一些小的代， 也意味着有 shingles 的字段会占用更多的磁 空 。 然而，大多数 用写入一次而取多次，所以在索引期 化我 的 速度是有意 的。

是一个在 Elasticsearch 里会 常 到的：不需要任何前期 行 多的 置，就能 在搜索的候有很好的效果。 一旦更清晰的理解了自己的需求，就能在索引 通 正 的 的数据建模 得更好果和性能。

部分匹配

敏 的 者会注意，目前 止本 介 的所有 都是 整个 的操作。 了能匹配，只能 倒排索引中存在的 ，最小的 元 个 。

但如果想匹配部分而不是全部的 ？ 部分匹配 允 用 指定 的一部分并 出所有包含部分片段的 。

与想象的不太一 ， 行部分匹配的需求在全文搜索引擎并不常 ，但是如果 者有 方面的背景，可能会在某个 候 一个低效的全文搜索用下面的 SQL 句 全文 行搜索：

```
WHERE text LIKE "%quick%"  
AND text LIKE "%brown%"  
AND text LIKE "%fox%" ①
```

① `fox` 会与 “fox” 和 “foxes” 匹配。

当然， Elasticsearch 提供分析 程，倒排索引 我 不需要使用 粗 的技 。 了能 同 匹配 “fox” 和 “foxes”的情况，只需 的将它 的 干作 索引形式，没有必要做部分匹配。

也就是 ， 在某些情况下部分匹配会比 有用，常 的 用如下：

- 匹配 、 品序列号或其他 `not_analyzed` 未分析 ， 些 可以是以某个特定前始，也可以是与某 模式匹配的，甚至可以是与某个正 式相匹配的。
- 入即搜索 (*search-as-you-type*) ——在用 入搜索 程的同 就呈 最可能的 果。
- 匹配如 或荷 有 合 的 言，如：*Weltgesundheitsorganisation* （世界 生， 英文 World Health Organization）。

本章始于 `not_analyzed` 精 字段的前 匹配。

与 化数据

我 会使用美国目前使用的 形式（United Kingdom postcodes 标准）来 明如何用部分匹配化数据。 形式有很好的 定 。例如， `W1V 3DG` 可以分解成如下形式：

- `W1V` : 是 的外部，它定 了 件的区域和行政区：
 - `W` 代表区域（1 或 2 个字母）

◦ **W1** 代表行政区（1或2个数字，可能跟着一个字符）

• **3DG**：内部定了街道或建筑：

- **3** 代表街区区（1个数字）
- **DG** 代表元（2个字母）

假设将作 **not_analyzed** 的精字段索引，所以可以为其建索引，如下：

```
PUT /my_index
{
  "mappings": {
    "address": {
      "properties": {
        "postcode": {
          "type": "string",
          "index": "not_analyzed"
        }
      }
    }
  }
}
```

然后索引一些：

```
PUT /my_index/address/1
{ "postcode": "W1V 3DG" }

PUT /my_index/address/2
{ "postcode": "W2F 8HW" }

PUT /my_index/address/3
{ "postcode": "W1F 7HW" }

PUT /my_index/address/4
{ "postcode": "WC1N 1LZ" }

PUT /my_index/address/5
{ "postcode": "SW5 0BE" }
```

在这些数据已可。

prefix 前

为了到所有以 **W1** 始的，可以使用 **prefix**：

```

GET /my_index/address/_search
{
  "query": {
    "prefix": {
      "postcode": "W1"
    }
  }
}

```

prefix 是一个 的底 的 ，它不会在搜索之前分析 字符串，它假定 入前 就正是要的前 。

TIP 状 下， **prefix** 不做相 度 分 算，它只是将所有匹配的文 返回，并 条果 予 分 1。它的行 更像是 器而不是 。 **prefix** 和 **prefix** 器者 的区 就是 器是可以被 存的，而 不行。

之前已 提：“只能在倒排索引中 到存在的 ”，但我 并没有 些 的索引 行特殊 理， 个是以它 精 的方式存在于 个文 的索引中，那 **prefix** 是如何工作的 ？

回想倒排索引包含了一个有序的唯一 列表（本例是 ）。 于 个 ，倒排索引都会将包含 的文 ID 列入倒排表（*postings list*）。与示例 的倒排索引是：

Term:	Doc IDs:
"SW5 0BE"	5
"W1F 7HW"	3
"W1V 3DG"	1
"W2F 8HW"	2
"WC1N 1LZ"	4

了支持前 匹配， 会做以下事情：

1. 描 列表并 到第一个以 **W1** 始的 。
2. 搜集 的文 ID 。
3. 移 到下一个 。
4. 如果 个 也是以 **W1** ， 跳回到第二 再重 行，直到下一个 不以 **W1** 止。

于小的例子当然可以正常工作，但是如果倒排索引中有数以百万的 都是以 **W1** ， 前 需要 个 然后 算 果！

前 越短所需 的 越多。如果我 要以 **W** 作 前 而不是 **W1** ，那 就可能需要做千万次的匹配。

CAUTION **prefix** 或 于一些特定的匹配是有效的，但使用方式 是 当注意。当字段中 的集合很小 ，可以放心使用，但是它的伸 性并不好，会 我 的集群 来很多 力。可以使用 的前 来限制 影 ， 少需要 的量。

本章后面会介绍一个索引的解决方案，一个方案能使前匹配更高效，不在此之前，需要先看看一个相似的：`wildcard` 和 `regexp`（模糊和正式）。

通配符与正表式

与 `prefix` 前的特性类似，`wildcard` 通配符也是一底基于的，与前不同的是它允许指定匹配的正式。它使用标准的 shell 通配符：`? 匹配任意字符，* 匹配 0 或多个字符。`

一个会匹配包含 `W1F 7HW` 和 `W2F 8HW` 的文：

```
GET /my_index/address/_search
{
  "query": {
    "wildcard": {
      "postcode": "W?F*HW" ①
    }
  }
}
```

① `? 匹配 1 和 2，* 与空格及 7 和 8 匹配。`

想如果只想匹配 `W` 区域的所有，前匹配也会包括以 `WC` 的所有，与通配符匹配到的相似，如果想匹配只以 `W` 始并跟随一个数字的所有，`regexp 正式` 允写出更的模式：

```
GET /my_index/address/_search
{
  "query": {
    "regexp": {
      "postcode": "W[0-9].+" ①
    }
  }
}
```

① 一个正表式要求必以 `W`，跟 0 至 9 之任何一个数字，然后接一或多个其他字符。

`wildcard` 和 `regexp` 的工作方式与 `prefix` 完全一，它也需要倒排索引中的列表才能到所有匹配的，然后依次取一个相的文 ID，与 `prefix` 的唯一不同是：它能支持更的匹配模式。

这意味着需要同注意前存在性能，有很多唯一的字段行些可能会消耗非常多的源，所以要避免使用左通配的模式匹配（如：`*foo` 或 `.*foo` 的正式）。

数据在索引的理有助于提高前匹配的效率，而通配符和正表式只能在完成，尽管些有其用景，但使用需慎。

`prefix`、`wildcard` 和 `regexp` 是基于 `操作的`，如果用它 来 `analyzed` 字段，它 会 `字段里面的 个`，而不是将字段作 整体来 理。

比方 包含 “Quick brown fox”（快速的棕色狐狸）的 `title` 字段会生成： `quick`、`brown` 和 `fox`。

会匹配以下 个：

```
{ "regexp": { "title": "br.*" }}
```

CAUTION

但是不会匹配以下 个：

```
{ "regexp": { "title": "Qu.*" } } ①  
{ "regexp": { "title": "quick br*" } } ②
```

① 在索引里的 是 `quick` 而不是 `Quick`。

② `quick` 和 `brown` 在 表中是分 的。

入即搜索

把 的事情先放一，我 先看看前 是如何在全文 中起作用的。用 已 在 完 内容之前，就能 他 展 搜索 果，就是所 的 即 搜索 (*instant search*) 或 入即搜索 (*search-as-you-type*) 。不 用 能在更短的 内得到搜索 果，我 也能引 用 搜索索引中真 存在的 果。

例如，如果用 入 `johnnie walker bl`，我 希望在它 完成 入搜索条件前就能得到：Johnnie Walker Black Label 和 Johnnie Walker Blue Label。

生活 是 ，就像猫的花色 不只一！我 希望能 到一 最 的 方式。并不需要 数据做任何准，在 就能 任意的全文字段 入即搜索 (*search-as-you-type*) 的 。

在 短 匹配 中，我 引入了 `match_phrase` 短 匹配 ，它匹配相 序一致的所有指定 ，于 的 入即搜索，可以使用 `match_phrase` 的一 特殊形式，`match_phrase_prefix`：

```
{  
  "match_phrase_prefix": {  
    "brand": "johnnie walker bl"  
  }  
}
```

的行 与 `match_phrase` 一致，不同的是它将 字符串的最后一个 作 前 使用，句 ，可以将之前的例子看成如下：

- `johnnie`
- 跟着 `walker`

- 跟着以 `bl` 始的

如果通过 `validate-query API` 行一个 , explanation 的结果 :

```
"johnnie walker bl*"
```

与 `match_phrase` 一样，它也可以接受 `slop` 参数（参照 `slop`）相 序位置不严格：

```
{
  "match_phrase_prefix": {
    "brand": {
      "query": "walker johnnie bl", ①
      "slop": 10
    }
  }
}
```

① 尽管 序不正，然能匹配，因 我 它 置了足 高的 `slop` 使匹配 的序有更大的 活性。

但是只有 字符串的最后一个 才能当作前 使用。

在之前的 前 中，我 警告 使用前 的 ，即 `prefix` 存在 重的 源消耗 ，短的方式也同 如此。前 `a` 可能会匹配成千上万的 ， 不 会消耗很多系 源，而且 果的用也不大。

可以通 置 `max_expansions` 参数来限制前 展的影 ，一个合理的 是可能是 50 :

```
{
  "match_phrase_prefix": {
    "brand": {
      "query": "johnnie walker bl",
      "max_expansions": 50
    }
  }
}
```

参数 `max_expansions` 控制着可以与前 匹配的 的数量，它会先 第一个与前 `bl` 匹配的，然后依次 搜集与之匹配的 （按字母 序），直到没有更多可匹配的 或当数量超 `max_expansions` 束。

不要忘 ，当用 多 入一个字符 ， 个 又会 行一遍，所以 需要快，如果第一个 果集不是用 想要的，他 会 入直到能搜出 意的 果 止。

索引 化

到目前 止，所有 的解决方案都是在 `(query time)` 的。

做并不需要特殊的映射或特殊的索引模式，只是 使用已 索引的数据。

的 活性通常会以 牺搜索性能 代 ，有 时候将 些消耗从 程中 移到 的地方是有意 的 。在 web 用中， 100 秒可能是一个 以忍受的巨大延 。

可以通 在索引 理数据提高搜索的 活性以及提升系 性能。此 然需要付出 有的代 加的索 引空 与 慢的索引能力，但 与 次 都需要付出代 不同，索引 的代 只用付出一次。

用 会感 我 。

Ngrams 在部分匹配的 用

之前提到：“只能在倒排索引中 到存在的 。” 尽管 `prefix` 、 `wildcard` 、 `regexp` 告 我 法并不完全正 ，但 个 的 要比在 列表中盲目挨个 效率要高得多。在搜索之前准 好供部分匹配的数据可以提高搜索的性能。

在索引 准 数据意味着要 合 的分析 ， 里部分匹配使用的工具是 *n-gram* 。可以将 *n-gram* 看成一个在 上滑 口， *n* 代表 个“ 口”的 度。如果我 要 n-gram `quick` 个 —— 它的 果取决于 *n* 的 度：

度 1 (unigram) : [`q`, `u`, `i`, `c`, `k`]

度 2 (bigram) : [`qu`, `ui`, `ic`, `ck`]

度 3 (trigram) : [`qui`, `uic`, `ick`]

度 4 (four-gram) : [`quic`, `uick`]

度 5 (five-gram) : [`quick`]

朴素的 n-gram 内部的匹配 非常有用，即在 Ngram 匹配 合 介 的那 。但 于 入即搜索 (search-as-you-type) 用 景，我 会使用一 特殊的 n-gram 称 界 n-grams (edge n-grams)。所 的 界 n-gram 是 它会固定 始的一 ，以 `quick` 例，它的 界 n-gram 的 果 :

- `q`
- `qu`
- `qui`
- `quic`
- `quick`

可能会注意到 与用 在搜索 入 “quick” 的字母次序是一致的， 句 ， 方式正好 足即 搜索 (instant search) !

索引 入即搜索

置索引 入即搜索的第一 是需要定 好分析 ，我 已在 配置分析器 中 ， 里会 些 再次 明。

准 索引

第一 需要配置一个自定 的 `edge_ngram` token 器，称 `autocomplete_filter`：

```
{  
  "filter": {  
    "autocomplete_filter": {  
      "type": "edge_ngram",  
      "min_gram": 1,  
      "max_gram": 20  
    }  
  }  
}
```

个配置的意思是：于 个 token 器接收的任意 ， 器会 之生成一个最小固定 1 , 最大 20 的 n-gram。

然后会在一个自定 分析器 `autocomplete` 中使用上面 个 token 器：

```
{  
  "analyzer": {  
    "autocomplete": {  
      "type": "custom",  
      "tokenizer": "standard",  
      "filter": [  
        "lowercase",  
        "autocomplete_filter" ①  
      ]  
    }  
  }  
}
```

① 自定 的 `edge-ngram` token 器。

个分析器使用 `standard` 分 器将字符串拆分 独立的 ， 并且将它 都 成小写形式，然后 个 生成一个 界 n-gram， 要感 `autocomplete_filter` 起的作用。

建索引、 例化 token 器和分析器的完整示例如下：

```

PUT /my_index
{
  "settings": {
    "number_of_shards": 1, ①
    "analysis": {
      "filter": {
        "autocomplete_filter": { ②
          "type": "edge_ngram",
          "min_gram": 1,
          "max_gram": 20
        }
      },
      "analyzer": {
        "autocomplete": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "autocomplete_filter" ③
          ]
        }
      }
    }
  }
}

```

① 参考 被破坏的相 度。

② 首先自定 token 器。

③ 然后在分析器中使用它。

可以拿 `analyze` API 一个新的分析器 保它行 正：

```

GET /my_index/_analyze?analyzer=autocomplete
quick brown

```

果表明分析器能正 工作，并返回以下：

- `q`
- `qu`
- `qui`
- `quic`
- `quick`
- `b`
- `br`
- `bro`
- `brow`

- brown

可以用 `update-mapping` API 将一个分析器用到具体字段：

```
PUT /my_index/_mapping/my_type
{
  "my_type": {
    "properties": {
      "name": {
        "type": "string",
        "analyzer": "autocomplete"
      }
    }
  }
}
```

在 建一些 文：

```
POST /my_index/my_type/_bulk
{ "index": { "_id": 1 } }
{ "name": "Brown foxes" }
{ "index": { "_id": 2 } }
{ "name": "Yellow furballs" }
```

字段

如果使用 `match` “brown fo”：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "name": "brown fo"
    }
  }
}
```

可以看到 个文 同 都能匹配，尽管 `Yellow furballs` 个文 并不包含 `brown` 和 `fo`：

```
{
  "hits": [
    {
      "_id": "1",
      "_score": 1.5753809,
      "_source": {
        "name": "Brown foxes"
      }
    },
    {
      "_id": "2",
      "_score": 0.012520773,
      "_source": {
        "name": "Yellow furballs"
      }
    }
  ]
}
```

如往常一，`validate-query` API 能提供一些 索：

```
GET /my_index/my_type/_validate/query?explain
{
  "query": {
    "match": {
      "name": "brown fo"
    }
  }
}
```

`explanation` 表明 会 界 n-grams 里的 个：

```
name:b name:br name:bro name:brow name:brown name:f name:fo
```

`name:f` 条件可以 足第二个文，因 `furballs` 是以 `f`、`fu`、`fur` 形式索引的。回 看 并不令人 相同的 `autocomplete` 分析器同 被 用于索引 和搜索， 在大多数情况下是正 的，只有在少数 景下才需要改 行。

我 需要保 倒排索引表中包含 界 n-grams 的 个，但是我 只想匹配用 入的完整 (`brown` 和 `fo`)，可以通 在索引 使用 `autocomplete` 分析器，并在搜索 使用 `standard` 准分析器来 想法，只要改 使用的搜索分析器 `analyzer` 参数即可：

```

GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "name": {
        "query": "brown fo",
        "analyzer": "standard" ①
      }
    }
  }
}

```

① 覆盖了 name 字段 analyzer 的配置。

这种方式，我可以在映射中，name 字段分指定 index_analyzer 和 search_analyzer。因 我只想改 search_analyzer，里只要更新 有的映射而不用 数据重新 建索引：

```

PUT /my_index/my_type/_mapping
{
  "my_type": {
    "properties": {
      "name": {
        "type": "string",
        "index_analyzer": "autocomplete", ①
        "search_analyzer": "standard" ②
      }
    }
  }
}

```

① 在索引，使用 autocomplete 分析器生成 索引 n-grams 的 个。

② 在搜索，使用 standard 分析器只搜索用 入的。

如果再次 求 validate-query API，当前的解：

```
name:brown name:fo
```

再次 行 就能正 返回 Brown foxes 个文。

因 大多数工作是在索引 完成的，所有的 只要 brown 和 fo 个，比使用 match_phrase_prefix 所有以 fo 始的 的方式要高效 多。

全提示 (Completion Suggester)

使用 界 n-grams 行 入即搜索 (search-as-you-type) 的 置 、 活且快速，但有 时候它并不 快，特 是当 立刻 得反 ，延 的 就会凸 ，很多 时候不搜索才是最快的 搜索方式。

Elasticsearch 里的 {ref}/search-suggesters-completion.html[completion suggester] 采用与上面完全不同的方式，需要 搜索条件生成一个所有可能完成的 列表，然后将它 置入一个 有限状 机 (*finite state transducer*) 内， 是个 化的 。 了搜索建 提示，Elasticsearch 从 的 始 着匹配路径一个字符一个字符地 行匹配，一旦它 于用 入的末尾，Elasticsearch 就会 所有可能 束的当前路径，然后生成一个建 列表。

本数据 存于内存中，能使前 非常快，比任何一 基于 的 都要快很多， 名字或品 牌的自 全非常 用，因 些 通常是以普通 序 的：用 “Johnny Rotten” 而不是 “Rotten Johnny”。

当 序不是那 容易被 ， 界 n-grams 比完成建 者 (Completion Suggester) 更合 。即使 不是所有猫都是一个花色，那 只猫的花色也是相当特殊的。

界 n-grams 与

界 n-gram 的方式可以用来 化的数据，比如 本章之前示例 中的 (postcode)。当然 postcode 字段需要 analyzed 而不是 not_analyzed，不 可以用 keyword 分 器来 理它，就好像他 是 not_analyzed 的一 。

keyword 分 器是一个非操作型分 器， 个分

TIP 器不做任何事情，它接收的任何字符串都会被原 出，因此它可以用来 理 not_analyzed 的字段 ，但 也需要其他的一些分析 ，如将字母 成小写。

下面示例使用 keyword 分 器将 成 token 流， 就能使用 界 n-gram token 器：

```
{
  "analysis": {
    "filter": {
      "postcode_filter": {
        "type": "edge_ngram",
        "min_gram": 1,
        "max_gram": 8
      }
    },
    "analyzer": {
      "postcode_index": { ①
        "tokenizer": "keyword",
        "filter": [ "postcode_filter" ]
      },
      "postcode_search": { ②
        "tokenizer": "keyword"
      }
    }
  }
}
```

① `postcode_index` 分析器使用 `postcode_filter` 将成界 n-gram 形式。

② `postcode_search` 分析器可以将搜索看成 `not_analyzed` 未分析的。

Ngrams 在合的用

最后，来看看 n-gram 是如何用于搜索合的言中的。的特点是它可以将多小合成一个大的合以表它准或的意。例如：

Aussprachewörterbuch

音字典 (Pronunciation dictionary)

Militärgeschichte

争史 (Military history)

Weißkopfseeadler

(White-headed sea eagle, or bald eagle)

Weltgesundheitsorganisation

世界生 (World Health Organization)

Rindfleischetikettierungsaufgabenübertragungsgesetz

法案考代理管牛和牛肉的的 (The law concerning the delegation of duties for the supervision of cattle marking and the labeling of beef)

有些人希望在搜索“Wörterbuch”(字典)的候，能在果中看到“Aussprachewörterbuch”(音字典)。同，搜索“Adler”()的候，能将“Weißkopfseeadler”()包括在果中。

理 言的一 方式可以用 {ref}/analysis-compound-word-tokenfilter.html[合 token 器 (compound word token filter)] 将 合 拆分成各自部分, 但 方式的 果 量依 于 合 字典的 量。

一 方式就是将所有的 用 n-gram 行 理, 然后搜索任何匹配的片段——能匹配的片段越多, 文 的相 度越大。

假 某个 n-gram 是一个 上的滑 口, 那 任 何 度的 n-gram 都可以遍 个 。我 既希望 足 的 拆分的 具有意 , 又不至于因 太 而生成 多的唯一 。一个 度 3 的 *trigram* 可能是一个不 的 始 :

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "trigrams_filter": {
          "type": "ngram",
          "min_gram": 3,
          "max_gram": 3
        }
      },
      "analyzer": {
        "trigrams": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": [
            "lowercase",
            "trigrams_filter"
          ]
        }
      }
    },
    "mappings": {
      "my_type": {
        "properties": {
          "text": {
            "type": "string",
            "analyzer": "trigrams" ①
          }
        }
      }
    }
  }
}
```

① text 字段用 trigrams 分析器索引它的内容, 里 n-gram 的 度是 3 。

使用 `analyze` API trigram 分析器 :

```
GET /my_index/_analyze?analyzer=trigrams  
Weißkopfseeadler
```

返回以下：

```
wei, eiß, ißk, ßko, kop, opf, pfs, fse, see, eea, ead, adl, dle, ler
```

索引前述示例中的 合 来：

```
POST /my_index/my_type/_bulk  
{ "index": { "_id": 1 }}  
{ "text": "Aussprachewörterbuch" }  
{ "index": { "_id": 2 }}  
{ "text": "Militärgeschichte" }  
{ "index": { "_id": 3 }}  
{ "text": "Weißkopfseeadler" }  
{ "index": { "_id": 4 }}  
{ "text": "Weltgesundheitsorganisation" }  
{ "index": { "_id": 5 }}  
{ "text": "Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz" }
```

“Adler”（ ）的搜索 化 三个 **adl**、**dle** 和 **ler**：

```
GET /my_index/my_type/_search  
{  
    "query": {  
        "match": {  
            "text": "Adler"  
        }  
    }  
}
```

正好与 “Weißkopfsee-adler” 相匹配：

```
{  
    "hits": [  
        {  
            "_id": "3",  
            "_score": 3.3191128,  
            "_source": {  
                "text": "Weißkopfseeadler"  
            }  
        }  
    ]  
}
```

似 “Gesundheit”（健康）可以与 “Welt-gesundheit-sorganisation” 匹配，同 也能与 “Militär-ges-chichte” 和 “Rindfleischetikettierungsüberwachungsaufgabenübertragungs-ges-etz” 匹配，因 它 同 都有 trigram 生成的 ges :

使用合 的 `minimum_should_match` 可以将 些奇怪的 果排除，只有当 trigram 最少匹配数 足要求 ，文 才能被 是匹配的：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "text": {
        "query": "Gesundheit",
        "minimum_should_match": "80%"
      }
    }
  }
}
```

有点像全文搜索中霰 式的策略，可能会 致倒排索引内容 多，尽管如此，在索引具有很多 合 的 言，或 之 没有空格的 言（如：泰 ），它 不失 一 通用且有效的方法。

技 可以用来提升 召回率 —— 搜索 果中相 的文 数。它通常会与其他技 一起使用，例如 shingles（参 [shingles 瓦片](#) ），以提高精度和 个文 的相 度 分。

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，在全文搜索引擎 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` 可以禁用位置，一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：

$$idf(t) = 1 + \log(\text{numDocs} / (\text{docFreq} + 1)) \quad ①$$

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：

$$\text{norm}(d) = 1 / \sqrt{\text{numTerms}} \quad ①$$

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` 字段也可以通过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 。

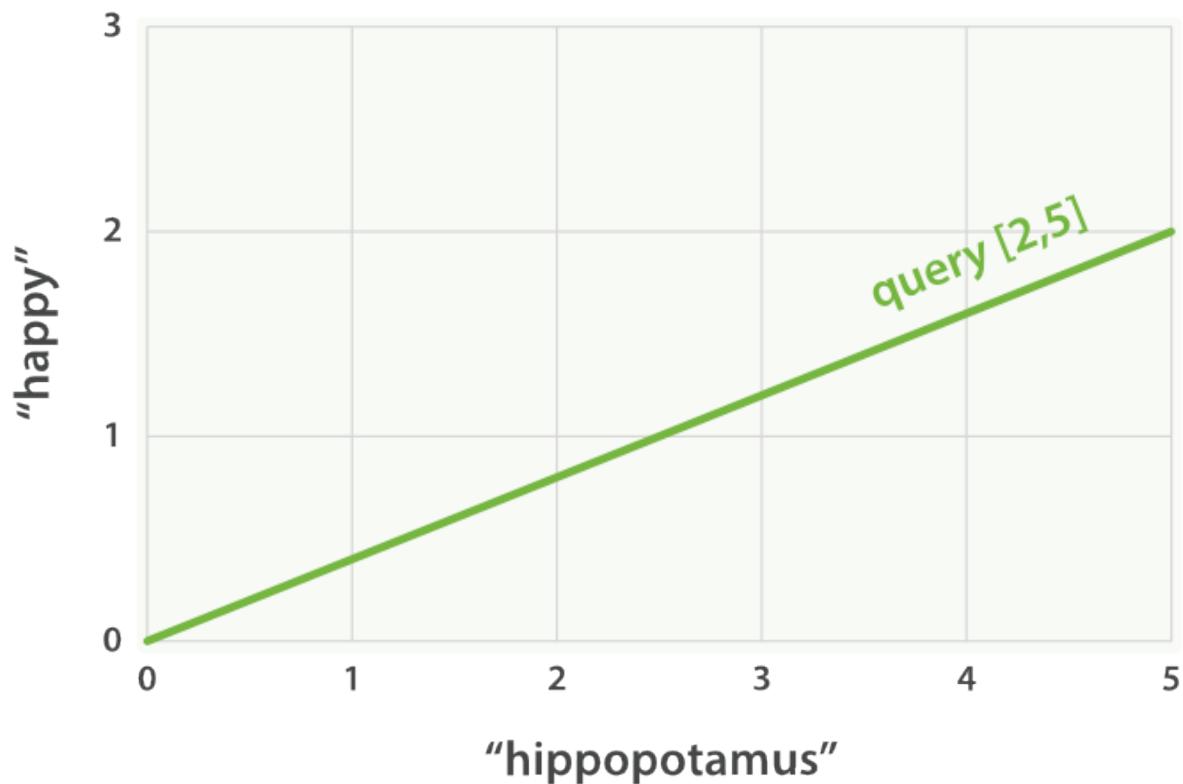


Figure 1. 表示 “*happy hippopotamus*” 的二向量

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]

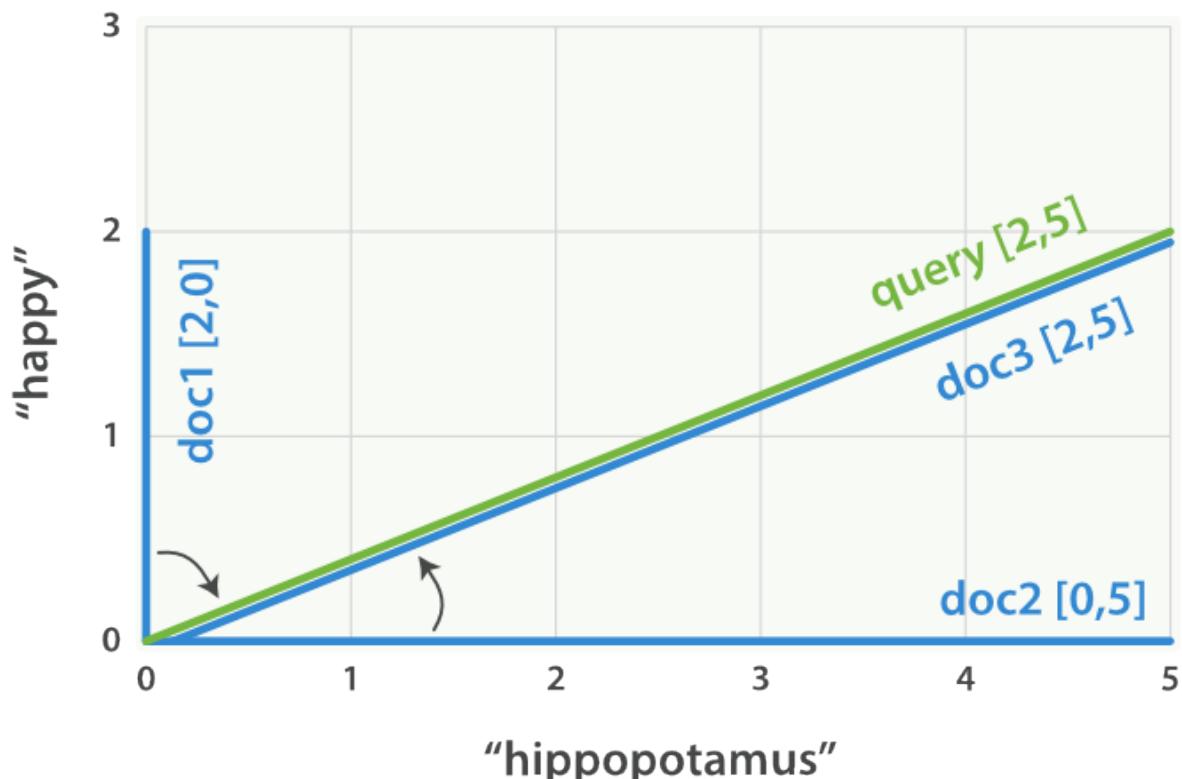


Figure 2. “happy hippopotamus” 及文 向量

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

```
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 布模型，在这个例子中，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
  ·  $\sum ($  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](#)）一起作一个字存于索引，`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 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 句的 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 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 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 纠争的 次， quick 、 fox 以及 red OR brown 是 于 且相互 纠争的。

我 已 知道 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 绍另外三个与相 度 分有 关的 方 法： boosting 、 constant_score 和 function_score 。

Not Quite Not

在互 联网上搜索 “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 等 的 ，从而将 果的 相 度 小至只返回与 “Apple” (果) 公司相 符的 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 `字段度一 (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` 来解决这个问题：

```
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，这个施都看成一个器，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，它允一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` 将主查询和函数包括在内。

② 主查询先行。

③ `field_value_factor` 函数会被用到一个与主 `query` 匹配的文档。

④ 每个文档的 `votes` 字段都必须有可供 `function_score` 算。如果没有文档的 `votes` 字段，那就必须使用 [{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，每个文档的最分 `_score` 都做了如下修改：

```
new_score = old_score * number_of_votes
```

然而并不会出人意料的好果，全文分 `_score` 通常于 0 到 10 之间，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。

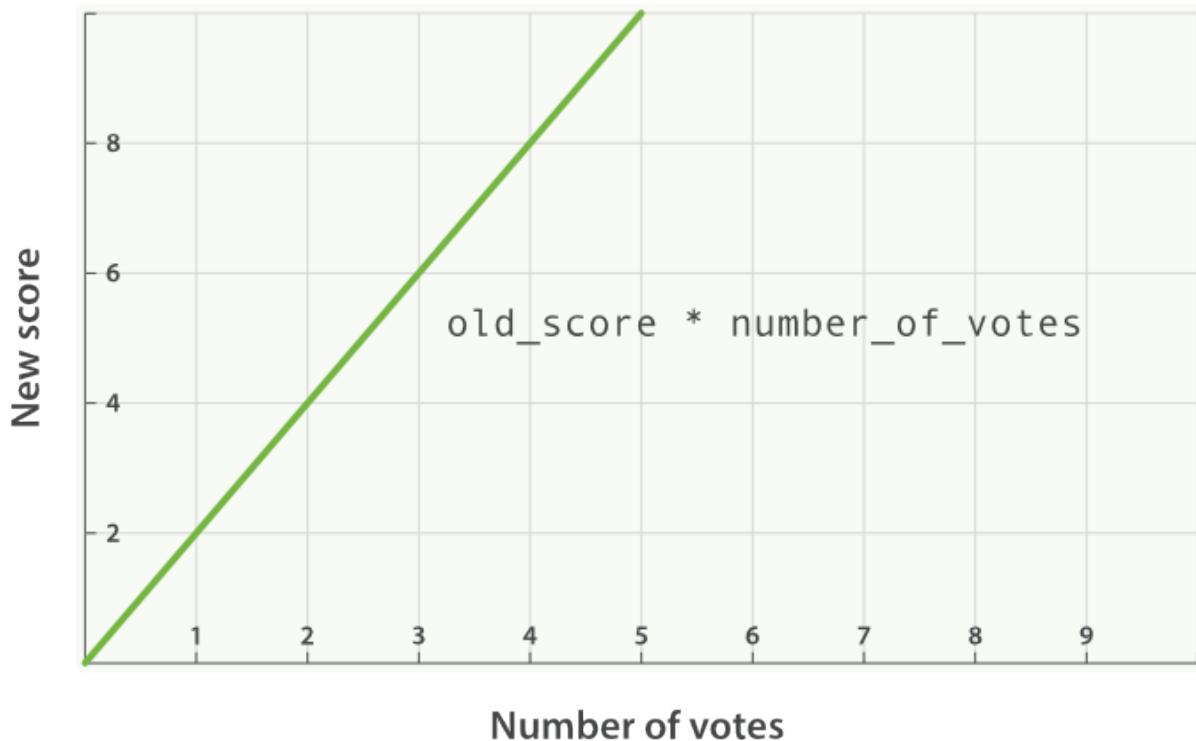


Figure 3. 受迎度的性系基于 `_score` 的原始 2.0

modifier

— 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：

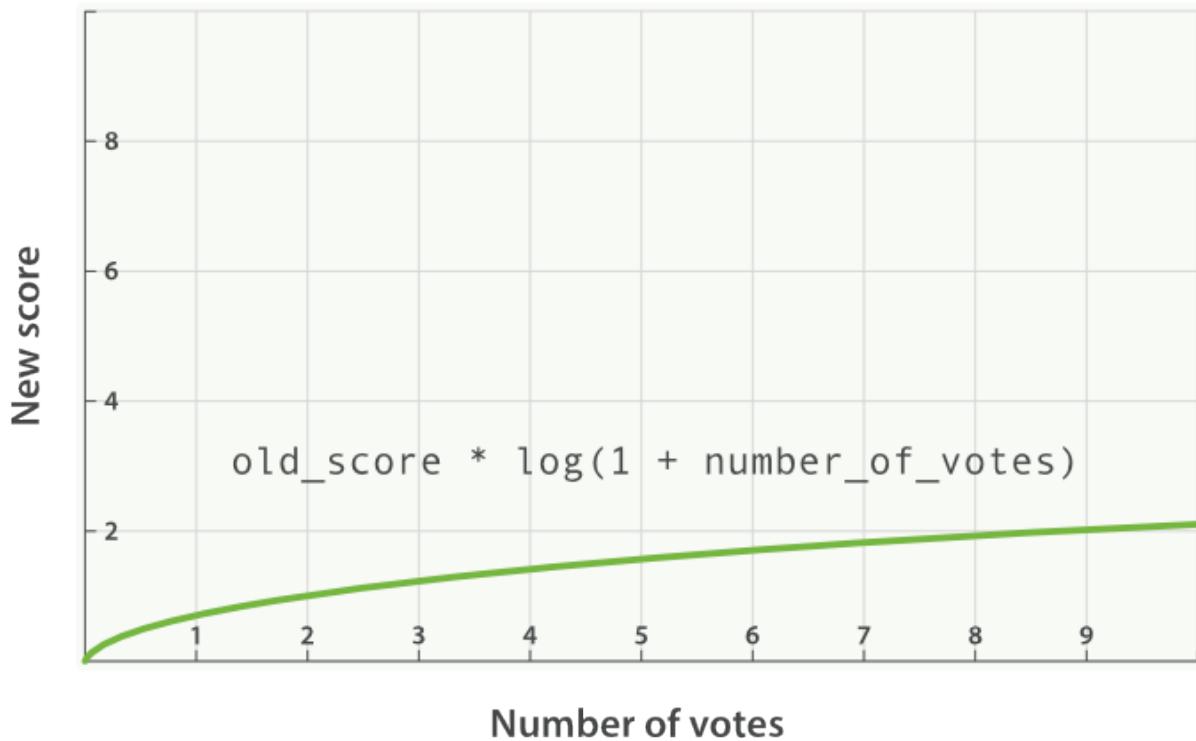


Figure 4. 受迎度的数系基于 `_score` 的原始 2.0

`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`。

修改 `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。

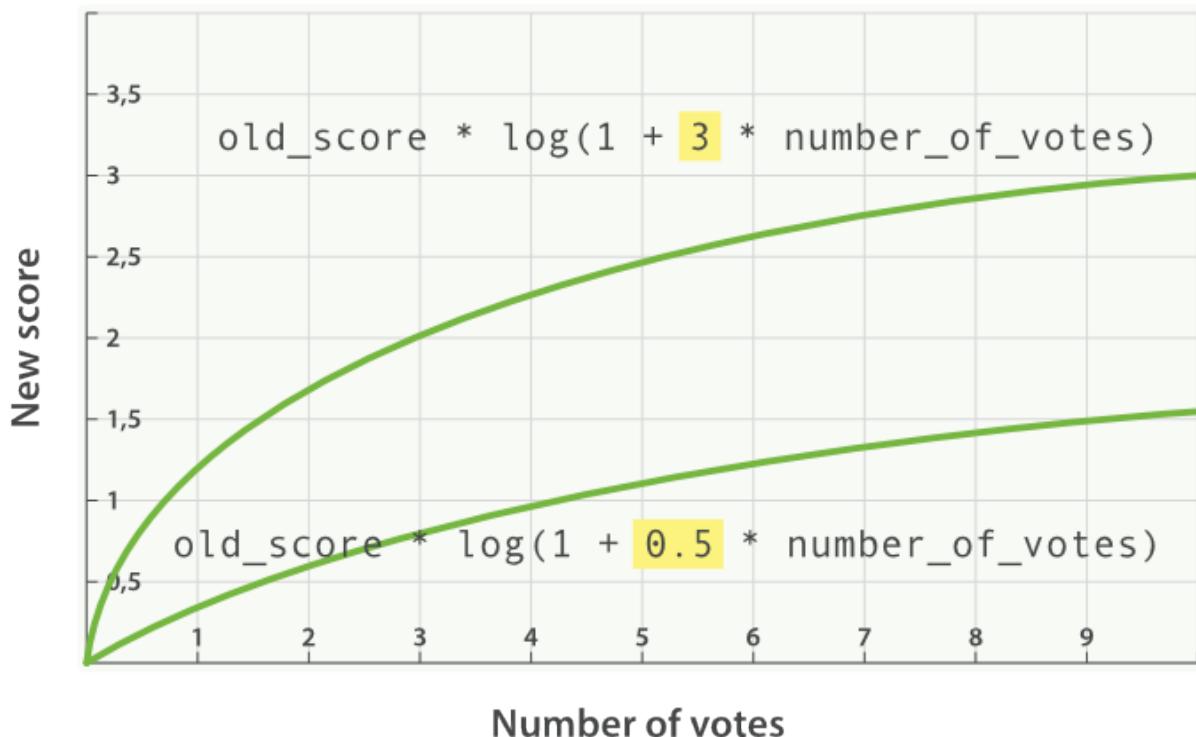


Figure 5. 受迎度的分数系基于多个不同因子

boost_mode

或全文分与 `field_value_factor` 函数乘的效果然可能太大，我可以通过参数 `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，特别是使用一个 `small_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)
```

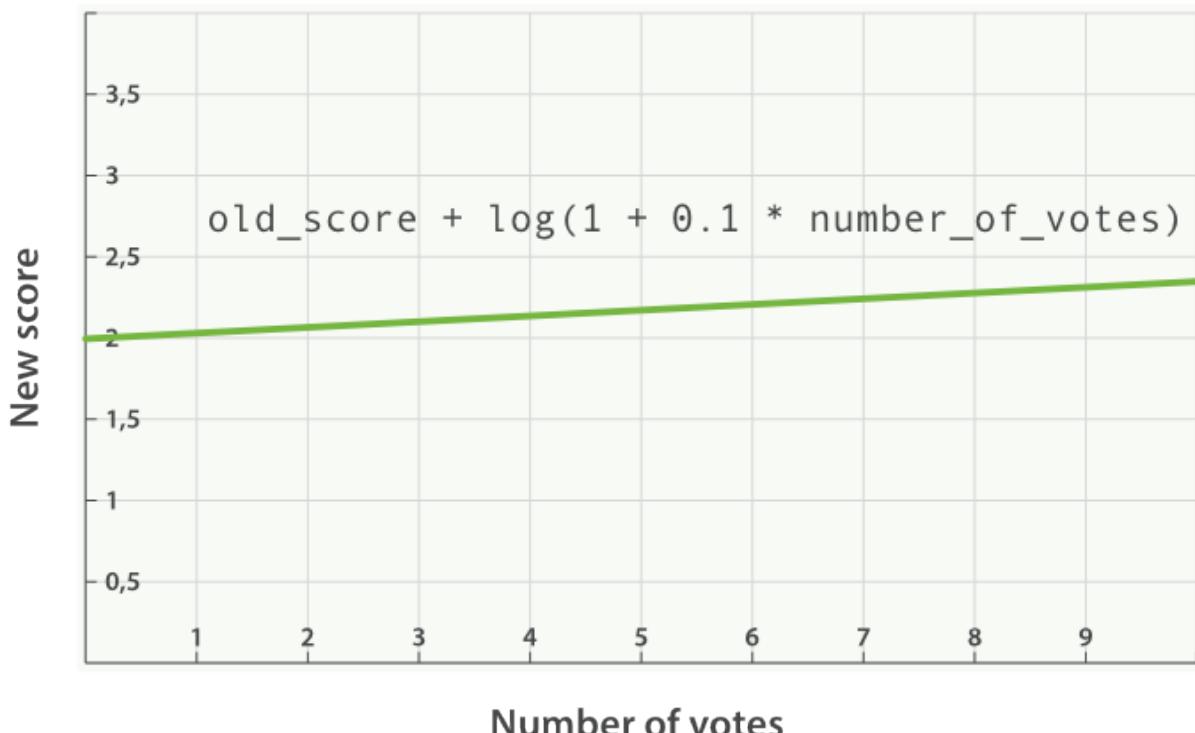


Figure 6. 使用 `sum` 合受 迎程度

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 user's 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 $-offset \leq origin \leq +offset$ 内的所有分 _score 都是 1.0。

三个函数的唯一区别就是它衰曲的形状，用来看会更直（参见衰函数曲）。

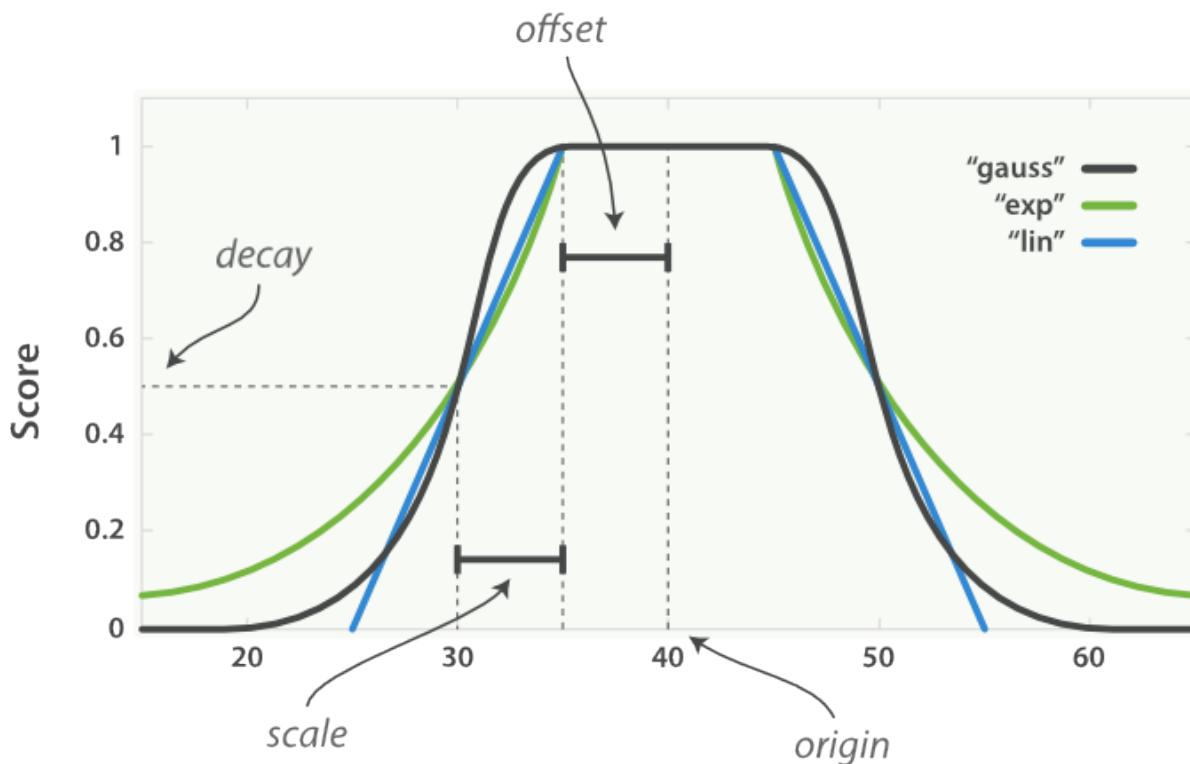


Figure 7. 衰函数曲

衰函数曲中所有曲线的原点 origin (即中心点) 的都是 40, offset 是 5，也就是在 $40 - 5 \leq value \leq 40 + 5$ 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 (线性、指数和高斯) 函数三者之的区别在于 (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 时候享受折扣 `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。

`script_score` 函数提供了巨大的活性，可以通过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响，如果脚本运行慢，可以有以下三：

TIP

- 尽可能多的提前算各信息并将结果存入一个文件中。
- 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]）。
- 那些最佳分的文件用脚本，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，这些算法可以参考{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，但这个算法也和 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显著影。但是如 TF/IDF 与 BM25 的和度所，文中共出 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*）。

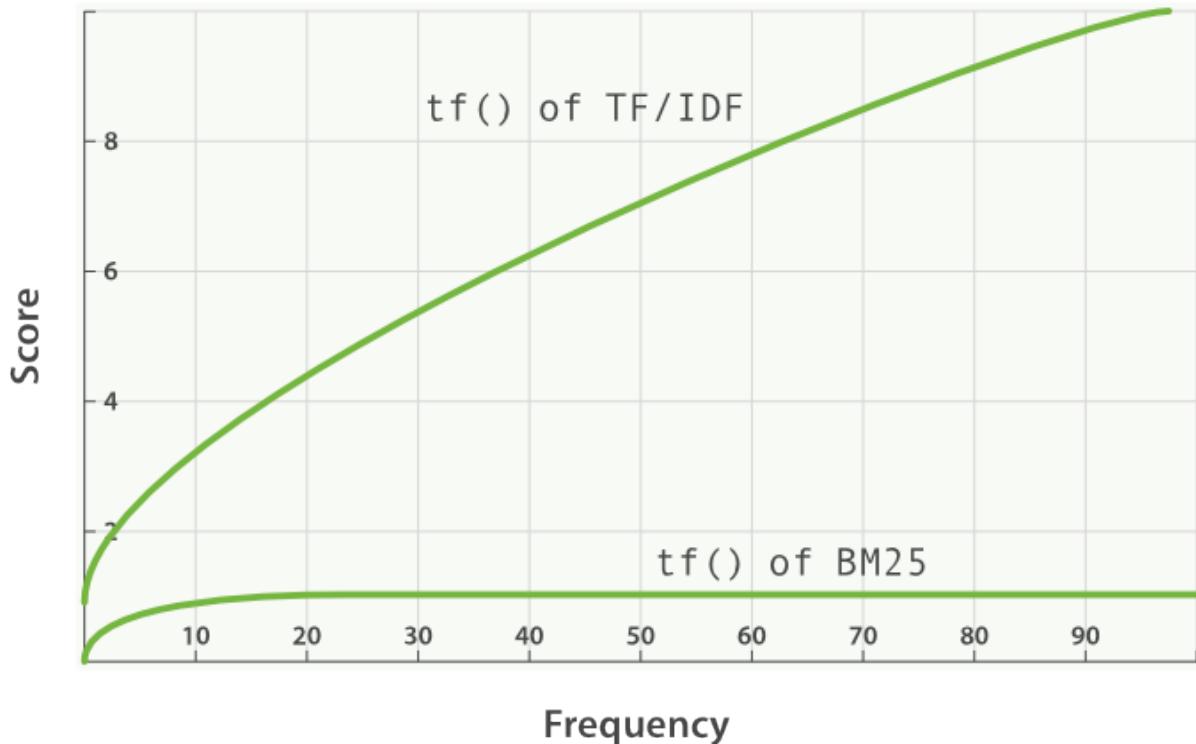


Figure 8. TF/IDF 与 BM25 的和度

字段度一化 (Field-length normalization)

在字段一化中，我提到Lucene会将短字段比长字段更重要：字段某个的度所来的重要性会被长字段的度抵消，但是分函数会将所有字段以同等方式对待。它所有短的title字段比所有的body字段更重要。

BM25当然也考虑短字段有更多的权重，但是它会参考各个字段内容的平均度，就能区分短title字段和长title字段。

CAUTION 在重提升中，已将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有一个更好的特性就是它提供了两个可参数：

k1

一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

一个参数控制着字段一的作用，**0.0**会禁用一化，**1.0**会用完全一化。**0.75**。

在实践中，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 反参数 行修改 。

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, 只能通过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
水平，唯一途径就是需要具 能 度量用 行 的 大能力。