写

我 期望在 似 和 格的 化数据上 行一个 来返回精 匹配的文 。 然而，好的全文索不 是完全相同的限定 。 相反，我 可以 大 以包括 可能 的匹配，而根据相 性得分将更好的匹配推到 果集的 部。

事 上，只能完全匹配的全文搜索可能会困 的用 。 道不希望在搜索 `quick brown fox` 匹配一个包含 `fast brown foxes` 的文 ， 搜索 `Johnny Walker` 同 匹配 `Johnnie Walker` ，搜索 `Arnold Shcwarzenneger` 同 匹配 `Arnold Schwarzenegger` ？

如果存在完全符合用 的文 ，他 出 在 果集的 部，而 弱的匹配可以被包含在列表的后面 。 如果没有精 匹配的文 ，至少我 可以 示有可能匹配用 要求的文 ，它 甚至可能是用 最初想要的！

我 已 在 [token-normalization] 看 自由 音匹配， [stemming] 中的 干， [synonyms] 中的同 ，但所有 些方法假定 写正 ，或者 个 写只有唯一的方法。

Fuzzy matching 允 匹配 写的 ，而 音 元 器可以在索引 用来 行 近似 音匹配。

模糊性

模糊匹配 待 “模糊” 相似的 个 似乎是同一个 。首先，我 需要 我 所 的 模糊性 行定 。

在1965年，Vladimir Levenshtein 出了 Levenshtein distance， 用来度量从一个 到一个 需要多少次 字符 。他提出了三 型的 字符 ：

- 一个字符 替 一个字符： _f_ox → _b_ox
- 入 一个新的字符：sic → sic_k_
- 除 一个字符：b_l_ack → back

Frederick Damerau 后来在 些操作基 上做了一个 展：

- 相 个字符的 位 ： _st_ar → _ts_ar

个例子，将 `bieber` 成 `beaver` 需要下面几个 ：

1. 把 `b` 替 成 `v` ：bie_b_er → bie_v_er
2. 把 `i` 替 成 `a` ：b_i_ever → b_a_ ever
3. 把 `e` 和 `a` 行 位：b_ae_ver → b_ea_ver

三个 表示 Damerau-Levenshtein edit distance 距 3。

然，从 <code>beaver</code> 成 <code>bieber</code> 是一个很 的 程—他 相距甚而不能 一个 的 写 。 Damerau 80% 的 写 距 1 。 句 ， 80% 的写 可以 原始字符串用 次 行修正。

Elasticsearch 指定了 `fuzziness` 参数支持 最大 距 的配置， 2 。

当然， 次 字符串的影 取决于字符串的 度。 hat 次 能 生 mad ， 所以 一个只有 3 个字符 度的字符串允 次 然太多了。 fuzziness 参数可以被 置 AUTO ， 将 致以下的最大 距 ：

- 字符串只有 1 到 2 个字符 是 0
- 字符串有 3 、 4 或者 5 个字符 是 1
- 字符串大于 5 个字符 是 2

当然， 可能会 距 2 然是太多了，返回的 果似乎并不相 。 把最大 fuzziness 置 1 ， 可以得到更好的 果和更好的性能。

模糊

{ref}/query-dsl-fuzzy-query.html[fuzzy]是 term 的模糊等 。 也 很少直接使用它，但是理解它是如何工作的，可以 助 在更高 的 match 中使用模糊性。

了解它是如何 作的，我 首先索引一些文 ：

```
POST /my_index/my_type/_bulk
{ "index": { "_id": 1 }}
{ "text": "Surprise me!"}
{ "index": { "_id": 2 }}
{ "text": "That was surprising."}
{ "index": { "_id": 3 }}
{ "text": "I wasn't surprised."}
```

在我 可以 surprize 行一个 fuzzy ：

```
GET /my_index/my_type/_search
{
  "query": {
    "fuzzy": {
      "text": "surprize"
    }
  }
}
```

fuzzy 是一个 的 ，所以它不做任何分析。它通 某个 以及指定的 fuzziness 到 典中所有的 。 fuzziness 置 AUTO 。

在我 的例子中， surprise 比 surprise 和 surprised 都在 距 2 以内， 所以文 1 和 3 匹配。通 以下 ，我 可以 少匹配度到 匹配 surprise ：

```
GET /my_index/my_type/_search
{
  "query": {
    "fuzzy": {
      "text": {
        "value": "surprize",
        "fuzziness": 1
      }
    }
  }
}
```

提高性能

<code>fuzzy</code> 的工作原理是 定原始 及 造一个 自 机— 像表示所有原始字符串指定 距 的字符串的一个大 表。

然后模糊 使用 个自 机依次高效遍 典中的所有 以 定是否匹配。 一旦收集了 典中存在的所有匹配 ，就可以 算匹配文 列表。

当然，根据存 在索引中的数据 型，一个 距 2 的模糊 能 匹配一个非常大数量的 同 行效率会非常糟 。下面 个参数可以用来限制 性能的影 ：

`prefix_length`
: 不能被`模糊化'' 的初始字符数。 大部分的 写 生在 的 尾，而不是 的 始。 例如通 将 `prefix_length 置 3， 可能 著降低匹配的 数量。

`max_expansions`
: 如果一个模糊 展了三个或四个模糊 ， 些新的模糊 也 是有意 的。如 果它 生 1000 个模糊 ，那 就基本没有意 了。 置 `max_expansions` 用来限制将 生的模糊 的 数量。模糊 将收集匹配 直到 到 `max_expansions` 的限制。

模糊匹配

`match` 支持 箱即用的模糊匹配：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "text": {
        "query":     "SURPRIZE ME!",
        "fuzziness": "AUTO",
        "operator":  "and"
      }
    }
  }
}
```

字符串首先 行分析，会 生 [surprize, me] ，并且 个 根据指定的 fuzziness 行模糊化。

同 ， multi_match 也支持 fuzziness ，但只有当 行 型是 best_fields 或者 most_fields ：

```
GET /my_index/my_type/_search
{
  "query": {
    "multi_match": {
      "fields":  [ "text", "title" ],
      "query":     "SURPRIZE ME!",
      "fuzziness": "AUTO"
    }
  }
}
```

match 和 multi_match 都支持 prefix_length 和 max_expansions 参数。

TIP	模糊性（Fuzziness）只能在 match and multi_match 中使用。不能使用在短 匹配、常用 或 cross_fields 匹配。

模糊性 分

用 喜 模糊 。他 会魔法般的 到正 写 合。很 憾， 效果平平。

假 我 有1000个文 包含 Schwarzenegger ，只是一个文 的出 写 Schwarzeneger 。 根据 term frequency/inverse document frequency 理 ， 个 写 文 比 写正 的相 度更高，因 写出 在更少的文 中！

句 ，如果我 待模糊匹配 似其他匹配方法，我 将偏 的 写超 了正 的 写， 会 用 狂。

TIP	模糊匹配不 用于参与 分—只能在有 写 大匹配 的 。

情况下， match 定所有的模糊匹配的恒定 分 1。 可以 足在果列表的末尾添加潜在的匹配 ，并且没有干 非模糊 的相 性 分。

TIP	在模糊 最初出 很少能 独使用。他 更好的作 一个 bigger 景的部分功能特性，如 *search-as-you-type* {ref}/search-suggesters-completion.html[完成 建]或 *did-you-mean* {ref}/search-suggesters-phrase.html[短 建]。

音匹配

最后，在 任何其他匹配方法都无效后，我 可以求助于搜索 音相似的 ，即使他 的 写不同。

有一些用于将 成 音 的算法。 Soundex 算法是 些算法的鼻祖， 而且大多数 音算法是

Soundex 的改 或者 版本，例如 Metaphone 和 Double Metaphone （ 展了除英 以外的其他 言的 音匹配）， Caverphone 算法匹配了新西 的名称， Beider-Morse 算法吸收了 Soundex 算法 了更好的匹配 和依地 名称， Kölner Phonetik 了更好的 理 。

得一提的是， 音算法是相当 的，他 初衷 的 言通常是英 或 。 限制了他 的 用性。不 ， 了某些明 的目 ，并与其他技 相 合， 音匹配能 作 一个有用的工具。

首先， 需要从 https://www.elastic.co/guide/en/elasticsearch/plugins/current/analysis-phonetic.html 取 音分析 件并在集群的 个 点安装， 然后重 个 点。

然后， 可以 建一个使用 音 元 器的自定 分析器，并 下面的方法：

```
PUT /my_index
{
  "settings": {
    "analysis": {
      "filter": {
        "dbl_metaphone": { ①
          "type":    "phonetic",
          "encoder": "double_metaphone"
        }
      },
      "analyzer": {
        "dbl_metaphone": {
          "tokenizer": "standard",
          "filter":    "dbl_metaphone" ②
        }
      }
    }
  }
}
```

① 首先，配置一个自定 `phonetic` 元 器并使用 `double_metaphone` 器。

② 然后在自定 分析器中使用自定 元 器。

在我 可以通 `analyze` API 来 行 ：

```
GET /my_index/_analyze?analyzer=dbl_metaphone
Smith Smythe
```

个 `Smith` 和 `Smythe` 在同一位置 生 个 元： `SM0` 和 `XMT` 。 通 分析器播放 `John` ， `Jon` 和 `Johnnie` 将 生 个 元 `JN` 和 `AN` ，而 `Jonathon` 生 元 `JN0N` 和 `ANTN` 。

音分析器可以像任何其他分析器一 使用。 首先映射一个字段来使用它，然后索引一些数据：

```
PUT /my_index/_mapping/my_type
{
  "properties": {
    "name": {
      "type": "string",
      "fields": {
        "phonetic": { ①
          "type":     "string",
          "analyzer": "dbl_metaphone"
        }
      }
    }
  }
}

PUT /my_index/my_type/1
{
  "name": "John Smith"
}

PUT /my_index/my_type/2
{
  "name": "Jonnie Smythe"
}
```

① name.phonetic 字段使用自定 dbl_metaphone 分析器。

可以使用 match 来 行搜索：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "name.phonetic": {
        "query": "Jahnnie Smeeth",
        "operator": "and"
      }
    }
  }
}
```

个 返回全部 个文 ，演示了如何 行 的 音匹配。 用 音算法 算 分是没有 的。音匹配的目的不是 了提高精度，而是要提高召回率—以 展足 的 来捕 可能匹配的文 。

通常更有意 的使用 音算法是在 索到 果后，由 一台 算机 行消 和后 理，而不是由人 用直接使用。