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 域并不常 ，但是如果 者有 SQL 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 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）
  - `1V` 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 将 作 `not_analyzed` 的精 字段索引，所以可以 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　些数据已可　　。

prefix 前

了　到所有以 W1　始的　　，可以使用　　的 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 一个索引 的解决方案， 个方案能使前 匹配更高效，不 在此之前，需要先看看 个相 的 ： wildcard 和 regexp （模糊和正 ）。

通配符与正 表 式

与 prefix 前 的特性 似， wildcard 通配符 也是一 底 基于 的 ，与前 不同的是它允 指定匹配的正 式。它使用 准的 shell 通配符 ： ? 匹配任意字符， * 匹配 0 或多个字符。

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。

想如果在只想匹配 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 个正表式要求必以 W，跟 0 至 9 之的任何一个数字，然后接一或多个其他字符。

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

也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。

CAUTION

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.*" }}
```

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 validate-query API 行 个 ，explanation 的解 果 ：

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

与 match_phrase 一 ，它也可以接受 slop 参数（参照 slop ） 相 序位置不那 格：

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

① 尽管 的 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　牲搜索性能　代　，有　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　的效率要高得多。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　　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 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 了 name 字段 analyzer 的 置。

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 界 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 候它并不 快，特 是当 立刻 得反 ，延 的 就会凸 ，很多 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 。

TIP

`keyword` 分 器是一个非操作型分 器， 个分 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）

Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz

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 “Aussprachewörtebuch”（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。