placeholder3

聚合

在 之前，本 致力于搜索。 通 搜索，如果我 有一个 并且希望 到匹配 个 的文 集，就好比在大海 。

通 聚合，我 会得到一个数据的概 。我 需要的是分析和 全套的数据而不是 个文 ：

- 在大海里有多少 ？
- 的平均 度是多少？
- 按照 的制造商来 分， 的 度中位 是多少？
- 月加入到海中的 有多少？

聚合也可以回答更加 微的 ：

- 最受 迎的 的制造商是什 ？
- 里面有 常的 ？

聚合允 我 向数据提出一些 的 。 然功能完全不同于搜索，但它使用相同的数据 。 意味着聚合的 行速度很快并且就像搜索一 几乎是 的。

告和 表 是非常 大的。 可以 示 的数据， 立即回 ，而不是 的数据 行 （需要一周 去 行的 *Hadoop* 任 ）， 的 告随着 的数据 化而 化，而不是 先 算的、 的和不相 的。

最后，聚合和搜索是一起的。 意味着 可以在 个 求里同 相同的数据 行搜索 / 和分析。并且由于聚合是在用 搜索的上下文里 算的， 不只是 示四星酒店的数量，而是 示匹配 条件的四星酒店的数量。

聚合是如此 大以至于 多公司已 数据分析建立了大型 Elasticsearch 集群。

高 概念

似于 DSL 表 式，聚合也有 可 合 的 法：独立 元的功能可以被混合起来提供 需要的自定 行 。 意味着只需要学 很少的基本概念，就可以得到几乎无尽的 合。

要掌握聚合， 只需要明白 个主要的概念：

桶（*Buckets*）

足特定条件的文 的集合

指 （*Metrics*）

桶内的文 行 算

就是全部了！ 个聚合都是一个或者多个桶和零个或者多个指 的 合。翻 成粗略的SQL 句来解 ：

```
SELECT COUNT(color) ①
FROM table
GROUP BY color ②
```

① `COUNT(color)` 相当于指 。

② `GROUP BY color` 相当于桶。

桶在概念上 似于 SQL 的分 （GROUP BY），而指 似于 `COUNT()` 、 `SUM()` 、 `MAX()` 等 方法。

我 深入 个概念 并且了解和 个概念相 的 西。

桶

桶 来 就是 足特定条件的文 的集合：

- 一个雇 属于 男性 桶或者 女性 桶
- 奥 巴尼属于 桶
- 日期2014-10-28属于 十月 桶

当聚合 始被 行， 个文 里面的 通 算来决定符合 个桶的条件。如果匹配到，文 将放入相 的桶并接着 行聚合操作。

桶也可以被嵌套在其他桶里面，提供 次化的或者有条件的 分方案。例如，辛辛那提会被放入俄亥俄州 个桶，而 整个 俄亥俄州桶会被放入美国 个桶。

Elasticsearch 有很多 型的桶，能 通 很多 方式来 分文 （ 、最受 迎的 、年 区 、地理位置等等）。其 根本上都是通 同 的原理 行操作：基于条件来 分文 。

指

桶能 我 分文 到有意 的集合，但是最 我 需要的是 些桶内的文 行一些指 的 算。分桶是一 到目的的手段：它提供了一 文 分 的方法来 我 可以 算感 趣的指 。

大多数 指 是 的数学 算（例如最小 、平均 、最大 ， 有 ）， 些是通 文 的 来 算。在 践中，指 能 算像平均薪 、最高出 格、95%的 延 的数据。

桶和指 的 合

聚合 是由桶和指 成的。 聚合可能只有一个桶，可能只有一个指 ，或者可能 个都有。也有可能有一些桶嵌套在其他桶里面。例如，我 可以通 所属国家来 分文 （桶），然后 算 个国家的平均薪酬（指 ）。

由于桶可以被嵌套，我 可以 非常多并且非常 的聚合：

1.通 国家 分文 （桶）

2.然后通 性 分 个国家（桶）

3.然后通 年 区 分 性 （桶）

4.最后， 个年 区 算平均薪酬（指 ）

最后将告 个 <国家， 性 ， 年 > 合的平均薪酬。所有的 些都在一个 求内完成并且只遍 一次数据！

聚合

我 可以用以下几 定 不同的聚合和它 的 法， 但学 聚合的最佳途径就是用 例来 明。一旦我 得了聚合的思想，以及如何合理地嵌套使用它 ，那 法就 得不那 重要了。

NOTE	聚合的桶操作和度量的完整用法可以在 {ref}/search-aggregations.html[Elasticsearch参考] 中 到。本章中会涵 其中很多内容，但在 完本章后 看它会有助于我 它的整体能力有所了解。

所以 我 先看一个例子。我 将会 建一些 汽 商有用的聚合，数据是 于汽 交易的信息： 型、制造商、 、何 被出 等。

首先我 批量索引一些数据：

```
POST /cars/transactions/_bulk
{ "index": {}}
{ "price" : 10000, "color" : "red", "make" : "honda", "sold" : "2014-10-28" }
{ "index": {}}
{ "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05" }
{ "index": {}}
{ "price" : 30000, "color" : "green", "make" : "ford", "sold" : "2014-05-18" }
{ "index": {}}
{ "price" : 15000, "color" : "blue", "make" : "toyota", "sold" : "2014-07-02" }
{ "index": {}}
{ "price" : 12000, "color" : "green", "make" : "toyota", "sold" : "2014-08-19" }
{ "index": {}}
{ "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05" }
{ "index": {}}
{ "price" : 80000, "color" : "red", "make" : "bmw", "sold" : "2014-01-01" }
{ "index": {}}
{ "price" : 25000, "color" : "blue", "make" : "ford", "sold" : "2014-02-12" }
```

有了数据， 始 建我 的第一个聚合。汽 商可能会想知道 个 色的汽 量最好，用聚合可以易得到 果，用 `terms` 桶操作：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "aggs" : { ①
        "popular_colors" : { ②
            "terms" : { ③
              "field" : "color"
            }
        }
    }
}
```

① 聚合操作被置于　　参数 aggs 之下（如果　　意，完整形式 aggregations 同　有效）。

② 然后，可以　聚合指定一个我　想要名称，本例中是： popular_colors 。

③ 最后，定　　个桶的　型 terms 。

聚合是在特定搜索　果背景下　行的，　　也就是　它只是　　求的　外一个　　参数（例如，使用 /_search　端点）。　聚合可以与　　，但我　会　些在　限定聚合的　　（Scoping　Aggregations）中来解决　个　　。

NOTE	可能会注意到我　将　size　　置成　0　。我　并不　心搜索　果的具体内容，所以将返回　数　置　0 来提高　　速度。　置 size: 0 与 Elasticsearch 1.x 中使用 count 搜索型等　。

然后我　　聚合定　一个名字，名字的　　取决于使用者，　　的　果会以我　定　的名字　　，用就可以解析得到的　果。

随后我　定　聚合本身，在本例中，我　定　了一个　　terms　桶。　　个　terms　桶会　　个到的唯一　　建新的桶。　因　我　告　它使用　color　字段，所以　terms　桶会　　个　色　　建新桶。

我　　行聚合并　看　果：

```
{
...
   "hits": {
      "hits": [] ①
   },
   "aggregations": {
      "popular_colors": { ②
         "buckets": [
            {
               "key": "red", ③
               "doc_count": 4 ④
            },
            {
               "key": "blue",
               "doc_count": 2
            },
            {
               "key": "green",
               "doc_count": 2
            }
         ]
      }
   }
}
```

① 因 我 置了 size 参数，所以不会有 hits 搜索 果返回。

② popular_colors 聚合是作 aggregations 字段的一部分被返回的。

③ 个桶的 key 都与 color 字段里 到的唯一 。它 会包含 doc_count 字段，告 我 包含 的文 数量。

④ 个桶的数量代表 色的文 数量。

包含多个桶， 个 一个唯一 色（例如： 或 ）。 个桶也包括 聚合 桶的所有文 的数量。例如，有四 色的 。

前面的 个例子完全是 行的：一旦文 可以被搜到，它就能被聚合。 也就意味着我 可以直接将聚合的 果源源不断的 入 形 ，然后生成 的 表 。 不久， 又 了一 色的 ，我 的 形就会立即 更新 色 的 信息。

！ 就是我 的第一个聚合！

添加度量指

前面的例子告 我 个桶里面的文 数量， 很有用。但通常，我 的 用需要提供更 的文 度量。例如， 色汽 的平均 格是多少？

了 取更多信息，我 需要告 Elasticsearch 使用 个字段， 算何 度量。 需要将度量 嵌套在桶内， 度量会基于桶内的文 算 果。

我　　汽　的例子加入 average 平均度量：

```
GET /cars/transactions/_search
{
   "size" : 0,
   "aggs": {
      "colors": {
         "terms": {
            "field": "color"
         },
         "aggs": { ①
            "avg_price": { ②
               "avg": {
                  "field": "price" ③
               }
            }
         }
      }
   }
}
```

① 度量新 aggs 。

② 度量指定名字： avg_price 。

③ 最后， price 字段定 avg 度量。

正如所 ，我 用前面的例子加入了新的 aggs 。 个新的聚合 我 可以将 avg 度量嵌套置于 terms 桶内。 上， 就 个 色生成了平均 格。

正如 色 的例子，我 需要 度量起一个名字（ avg_price ） 可以 后根据名字 取它的 。最后，我 指定度量本身（ avg ）以及我 想要 算平均 的字段（ price ）：

```
{
...
   "aggregations": {
      "colors": {
         "buckets": [
            {
               "key": "red",
               "doc_count": 4,
               "avg_price": { ①
                  "value": 32500
               }
            },
            {
               "key": "blue",
               "doc_count": 2,
               "avg_price": {
                  "value": 20000
               }
            },
            {
               "key": "green",
               "doc_count": 2,
               "avg_price": {
                  "value": 21000
               }
            }
         ]
      }
   }
...
}
```

① 中的新字段 avg_price 。

尽管 只 生很小改 ， 上我 得的数据是 了。之前，我 知道有四 色的 ， 在， 色 的平均 格是 $32，500 美元。 个信息可以直接 示在 表或者 形中。

嵌套桶

在我 使用不同的嵌套方案 ，聚合的力量才能真正得以 。 在前例中，我 以及看到如何将一个度量嵌入桶中，它的功能已 十分 大了。

但真正令人激 的分析来自于将桶嵌套 外一个桶 所能得到的 果。 在，我 想知道 个 色的汽 制造商的分布：

```
GET /cars/transactions/_search
{
   "size" : 0,
   "aggs": {
      "colors": {
         "terms": {
            "field": "color"
         },
         "aggs": {
            "avg_price": { ①
               "avg": {
                  "field": "price"
               }
            },
            "make": { ②
                "terms": {
                    "field": "make" ③
                }
            }
         }
      }
   }
}
```

① 注意前例中的 avg_price 度量 然保持原位。

② 一个聚合 make 被加入到了 color 色桶中。

③ 个聚合是 terms 桶，它会 个汽 制造商生成唯一的桶。

里 生了一些有趣的事。 首先，我 可能会 察到之前例子中的 avg_price 度量完全没有 化，在原来的位置。 一个聚合的 个 都可以有多个度量或桶， avg_price 度量告 我 色汽的平均 格。它与其他的桶和度量相互独立。

我 的 用非常重要，因 里面有很多相互 ，但又完全不同的度量需要收集。聚合使我 能 用一次数据 求 得所有的 些信息。

外一件 得注意的重要事情是我 新 的 个 make 聚合，它是一个 terms 桶（嵌套在 colors 、 terms 桶内）。 意味着它会 数据集中的 个唯一 合生成（ color 、 make ）元 。

我 看看返回的 （ 了 我 只 示部分 果）：

```
{
...
   "aggregations": {
      "colors": {
         "buckets": [
            {
               "key": "red",
               "doc_count": 4,
               "make": { ①
                  "buckets": [
                     {
                        "key": "honda", ②
                        "doc_count": 3
                     },
                     {
                        "key": "bmw",
                        "doc_count": 1
                     }
                  ]
               },
               "avg_price": {
                  "value": 32500 ③
               }
            },

...
}
```

① 正如期望的那 ，新的聚合嵌入在 个 色桶中。

② 在我 看 按不同制造商分解的 色下 信息。

③ 最 ，我 看到前例中的 avg_price 度量 然 持不 。

果告 我 以下几点：

- 色 有四 。
- 色 的平均 是 $32，500 美元。
- 其中三 是 Honda 本田制造，一 是 BMW 宝 制造。

最后的修改

我 回到 的原点，在 入新 之前， 我 的示例做最后一个修改， 个汽 生成商算最低和最高的 格：

```
GET /cars/transactions/_search
{
   "size" : 0,
   "aggs": {
      "colors": {
         "terms": {
            "field": "color"
         },
         "aggs": {
            "avg_price": { "avg": { "field": "price" }
            },
            "make" : {
                "terms" : {
                    "field" : "make"
                },
                "aggs" : { ①
                    "min_price" : { "min": { "field": "price"} }, ②
                    "max_price" : { "max": { "field": "price"} } ③
                }
            }
         }
      }
   }
}
```

① 我 需要 加 外一个嵌套的 aggs 。

② 然后包括 min 最小度量。

③ 以及 max 最大度量。

得到以下 出（只 示部分 果）：

```
{
...
   "aggregations": {
      "colors": {
         "buckets": [
            {
               "key": "red",
               "doc_count": 4,
               "make": {
                  "buckets": [
                     {
                        "key": "honda",
                        "doc_count": 3,
                        "min_price": {
                           "value": 10000 ①
                        },
                        "max_price": {
                           "value": 20000 ①
                        }
                     },
                     {
                        "key": "bmw",
                        "doc_count": 1,
                        "min_price": {
                           "value": 80000
                        },
                        "max_price": {
                           "value": 80000
                        }
                     }
                  ]
               },
               "avg_price": {
                  "value": 32500
               }
            },
...
```

① min 和 max 度量　在出　在　个汽　制造商（ make ）下面。

有了　个桶，我　可以　的　果　行　展并得到以下信息：

- 有四　色　。
- 色　的平均　是 $32，500 美元。
- 其中三　色　是 Honda 本田制造，一　是 BMW 宝　制造。
- 最便宜的　色本田　$10，000 美元。
- 最　的　色本田　$20，000 美元。

条形

聚合 有一个令人激 的特性就是能 十分容易地将它 成 表和 形。本章中， 我 正在通 示例数据来来完成各 各 的聚合分析，最 ，我 将会 聚合功能是非常 大的。

直方 histogram 特 有用。 它本 上是一个条形 ，如果有 建 表或分析 表 的 ，那 我会 无疑 的 里面有一些 表是条形 。 建直方 需要指定一个区 ，如果我 要 建一个直方 ，可以将 隔 20,000。 做将会在 个 $20,000 建一个新桶，然后文 会被分到 的桶中。

于 表 来 ，我 希望知道 个 区 内汽 的 量。我 会想知道 个 区 内汽 所 来的收入，可以通 个区 内已 汽 的 求和得到。

可以用 histogram 和一个嵌套的 sum 度量得到我 想要的答案：

```
GET /cars/transactions/_search
{
   "size" : 0,
   "aggs":{
      "price":{
         "histogram":{ ①
            "field": "price",
            "interval": 20000
         },
         "aggs":{
            "revenue": {
               "sum": { ②
                 "field" : "price"
               }
             }
         }
      }
   }
}
```

① histogram 桶要求 个参数：一个数 字段以及一个定 桶大小 隔。

② sum 度量嵌套在 个 区 内，用来 示 个区 内的 收入。

如我 所 ， 是 price 聚合 建的，它包含一个 histogram 桶。它要求字段的 型必 是数 型的同 需要 定分 的 隔 。 隔 置 20,000 意味着我 将会得到如 [0-19999, 20000-39999, …] 的区 。

接着，我 在直方 内定 嵌套的度量， 个 sum 度量，它会 落入某一具体 区 的文 中 price 字段的 行求和。 可以 我 提供 个 区 的收入，从而可以 到底是普通家用 是奢 。

果如下：

```
{
...
   "aggregations": {
      "price": {
         "buckets": [
            {
               "key": 0,
               "doc_count": 3,
               "revenue": {
                  "value": 37000
               }
            },
            {
               "key": 20000,
               "doc_count": 4,
               "revenue": {
                  "value": 95000
               }
            },
            {
               "key": 80000,
               "doc_count": 1,
               "revenue": {
                  "value": 80000
               }
            }
         ]
      }
   }
}
```

果很容易理解，不　　注意到直方　的　　是区　的下限。　　0　代表区　　0-19, 999　，　　20000　代表区　20, 000-39, 999，等等。

NOTE

我　可能会注意到空的区　，比如：$40，000-60，000，没有出　在　　中。　histogram　桶　　会忽略它，因　它有可能会　致不希望的潜在　　出。

我　会在下一小　中　　如何包括空桶。返回空桶 返回空 Buckets 。

可以在　Sales and Revenue per price bracket 中看到以上数据直方　的　形化表示。

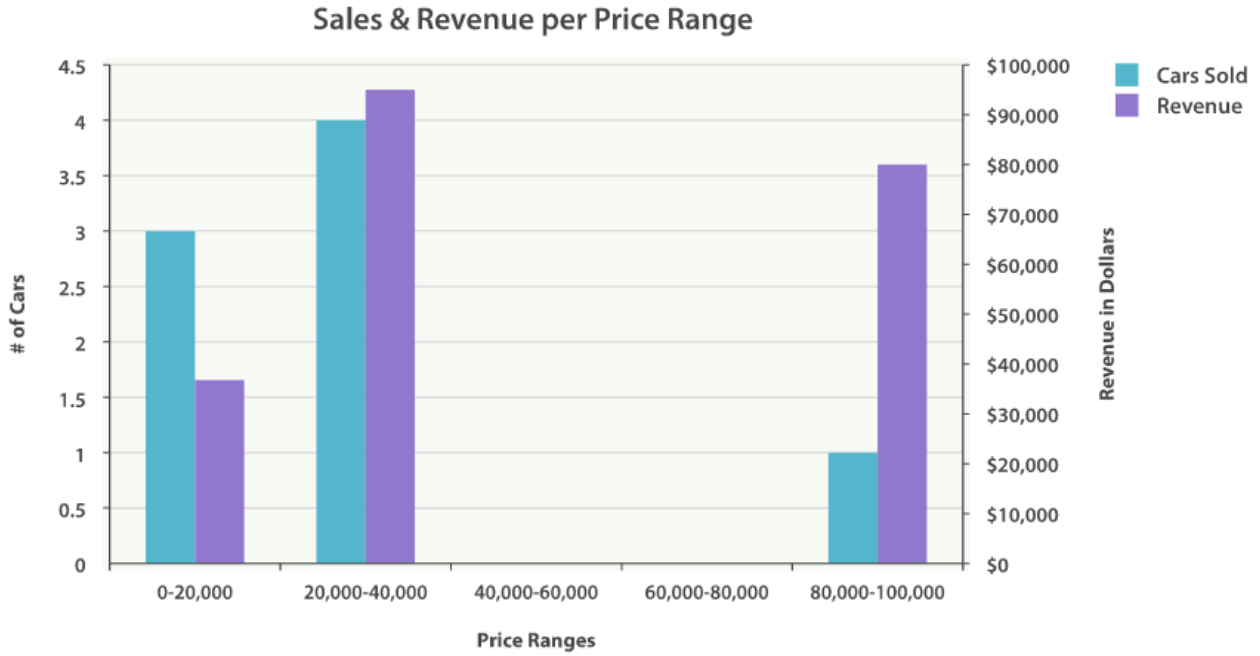

Figure 1. Sales and Revenue per price bracket

当然，我 可以 任何聚合 出的分 和 果 建条形 ，而不只是 直方 桶。 我 以最受 迎 10 汽 以及它 的平均 、 准差 些信息 建一个条形 。 我 会用到 terms 桶和 extended_stats 度量：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs": {
    "makes": {
      "terms": {
        "field": "make",
        "size": 10
      },
      "aggs": {
        "stats": {
          "extended_stats": {
            "field": "price"
          }
        }
      }
    }
  }
}
```

上述代 会按受 迎度返回制造商列表以及它 各自的 信息。我 其中的 stats.avg 、 stats.count 和 stats.std_deviation 信息特 感 趣，并用 它 算出 准差：

```
std_err = std_deviation / count
```

建 表如 Average price of all makes, with error bars 。

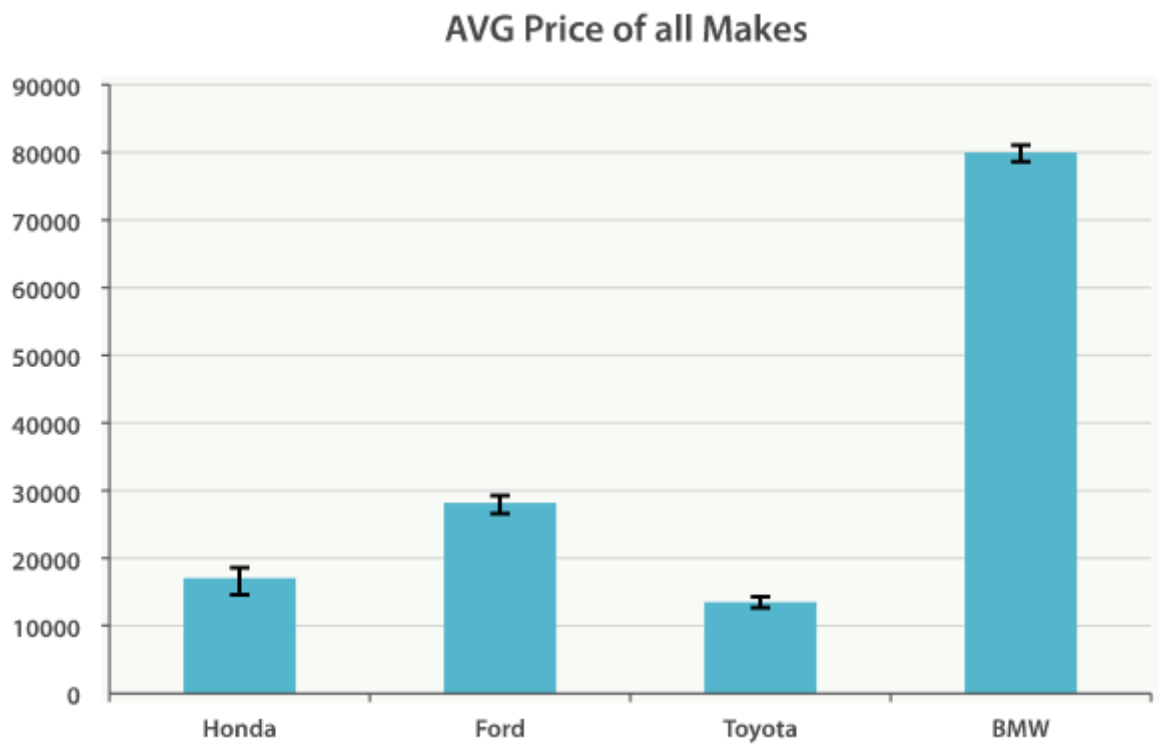

Figure 2. Average price of all makes, with error bars

按

如果搜索是在 Elasticsearch 中使用 率最高的，那 建按 的 date_histogram 随其后。什 会想用 date_histogram ？

假 的数据 。 无 是什 数据（Apache 事件日志、股票 交易 、棒球 ）只要有 都可以 行 date_histogram 分析。当 的数据有 ， 是想在 度上 建指分析：

- 今年 月 多少台汽 ？
- 只股票最近 12 小 的 格是多少？
- 我 站上周 小 的平均 延 是多少？

然通常的 histogram 都是条形 ，但 date_histogram 向于 成 状 以展示 序列。 多公司用 Elasticsearch 只是 了分析 序列数据。`date_histogram` 分析是它 最基本的需要。

`date_histogram` 与 通常的 `histogram` 似。 但不是在代表数 的数 字段上 建 buckets ，而是在 上 建 buckets。 因此 一个 bucket 都被定 成一个特定的日期大小 (比如， `1个月` 或 `2.5 天`)。

可以用通常的 **histogram** 行 分析 ？

从技 上来 ，是可以的。 通常的 `histogram` bucket（桶）是可以 理日期的。但是它不能自 日期。 而用 `date_histogram` ， 可以指定 段如 `1 个月` ，它能 明地知道 2 月的天数比 12 月少。 `date_histogram` 具有 外一个 ，即能合理地 理 区， 可以使用客 端的 区 行 定制，而不是用服 器端 区。

通常的 histogram 会把日期看做是数字， 意味着 必 以微秒 位指明 隔。 外聚合并不知道日 隔，使得它 于日期而言几乎没什 用 。

我 的第一个例子将 建一个 的折 来回答如下 ： 月 多少台汽 ？

```
GET /cars/transactions/_search
{
   "size" : 0,
   "aggs": {
      "sales": {
         "date_histogram": {
            "field": "sold",
            "interval": "month", ①
            "format": "yyyy-MM-dd" ②
         }
      }
   }
}
```

① 隔要求是日 (如 个 bucket 1 个月)。

② 我 提供日期格式以便 buckets 的 便于 。

我 的 只有一个聚合， 月 建一个 bucket。 我 可以得到 个月 的汽 数量。 外 提供了一个 外的 `format` 参数以便 buckets 有 "好看的" 。 然而在内部，日期 然是被表示成数 。 可能会使得 UI 者抱怨，因此可以提供常用的日期格式 行格式化以更方便 。

果既符合 期又有一点出人意料（看看 是否能 到意外之 ）：

```
{
   ...
   "aggregations": {
      "sales": {
         "buckets": [
            {
               "key_as_string": "2014-01-01",
               "key": 1388534400000,
               "doc_count": 1
            },
            {
               "key_as_string": "2014-02-01",
               "key": 1391212800000,
               "doc_count": 1
            },
            {
               "key_as_string": "2014-05-01",
               "key": 1398902400000,
               "doc_count": 1
            },
            {
               "key_as_string": "2014-07-01",
               "key": 1404172800000,
               "doc_count": 1
            },
            {
               "key_as_string": "2014-08-01",
               "key": 1406851200000,
               "doc_count": 1
            },
            {
               "key_as_string": "2014-10-01",
               "key": 1412121600000,
               "doc_count": 1
            },
            {
               "key_as_string": "2014-11-01",
               "key": 1414800000000,
               "doc_count": 2
            }
         ]
...
}
```

聚合 果已 完全展示了。正如 所 ，我 有代表月 的 buckets， 个月的文 数目，以及美化后的 `key_as_string` 。

返回空 Buckets

注意到 果末尾 的奇怪之 了 ？

是的， 果没 。 我 的 果少了一些月 ！ date_histogram （和 histogram 一 ） 只会返回文 数目非零的 buckets。

意味着 的 histogram 是返回最少 果。通常， 并不想要 。 于很多 用， 可能想直接把果 入到 形 中，而不想做任何后期加工。

事 上，即使 buckets 中没有文 我 也想返回。可以通 置 个 外参数来 效果：

```
GET /cars/transactions/_search
{
   "size" : 0,
   "aggs": {
      "sales": {
         "date_histogram": {
            "field": "sold",
            "interval": "month",
            "format": "yyyy-MM-dd",
            "min_doc_count" : 0, ①
            "extended_bounds" : { ②
                "min" : "2014-01-01",
                "max" : "2014-12-31"
            }
         }
      }
   }
}
```

① 个参数 制返回空 buckets。

② 个参数 制返回整年。

个参数会 制返回一年中所有月 的 果，而不考 果中的文 数目。 min_doc_count 非常容易理解：它 制返回所有 buckets，即使 buckets 可能 空。

extended_bounds 参数需要一点解 。 min_doc_count 参数 制返回空 buckets，但是 Elasticsearch 只返回 的数据中最小 和最大 之 的 buckets。

因此如果 的数据只落在了 4 月和 7 月之 ，那 只能得到 些月 的 buckets（可能 空也可能不空）。因此 了得到全年数据，我 需要告 Elasticsearch 我 想要全部 buckets， 即便那些 buckets 可能落在最小日期 之前 或 最大日期 之后 。

extended_bounds 参数正是如此。一旦 加上了 个 置， 可以把得到的 果 易地直接 入到 的 形 中，从而得到 似 汽 的 表。

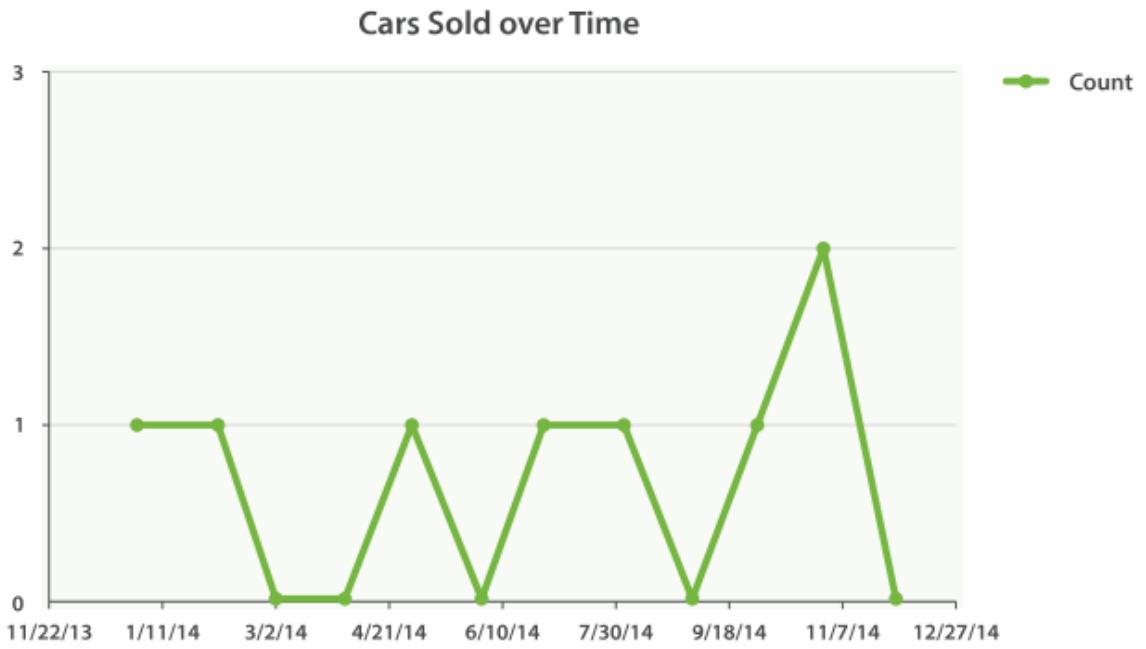

Figure 3. 汽

展例子

正如我 已 很多次，buckets 可以嵌套 buckets 中从而得到更 的分析。 作 例子，我 建聚合以便按季度展示所有汽 品牌 。同 按季度、按 个汽 品牌 算 ，以便可以出 品牌最 ：

```
GET /cars/transactions/_search
{
   "size" : 0,
   "aggs": {
      "sales": {
         "date_histogram": {
            "field": "sold",
            "interval": "quarter", ①
            "format": "yyyy-MM-dd",
            "min_doc_count" : 0,
            "extended_bounds" : {
                "min" : "2014-01-01",
                "max" : "2014-12-31"
            }
         },
         "aggs": {
            "per_make_sum": {
               "terms": {
                  "field": "make"
               },
               "aggs": {
                  "sum_price": {
                     "sum": { "field": "price" } ②
                  }
               }
            },
            "total_sum": {
               "sum": { "field": "price" } ③
            }
         }
      }
   }
}
```

① 注意我　把　　隔从 `month` 改成了 `quarter` 。

②　算　品牌的　　金　。

③ 也　算所有全部品牌的　　金　。

得到的　果（截去了一大部分）如下：

```
{
....
"aggregations": {
   "sales": {
      "buckets": [
         {
            "key_as_string": "2014-01-01",
            "key": 1388534400000,
            "doc_count": 2,
            "total_sum": {
               "value": 105000
            },
            "per_make_sum": {
               "buckets": [
                  {
                     "key": "bmw",
                     "doc_count": 1,
                     "sum_price": {
                        "value": 80000
                     }
                  },
                  {
                     "key": "ford",
                     "doc_count": 1,
                     "sum_price": {
                        "value": 25000
                     }
                  }
               ]
            }
         },
...
}
```

我 把 果 成 ，得到如 按品牌分布的 季度 所示的 的折 和 个品牌（ 季度）的柱状 。

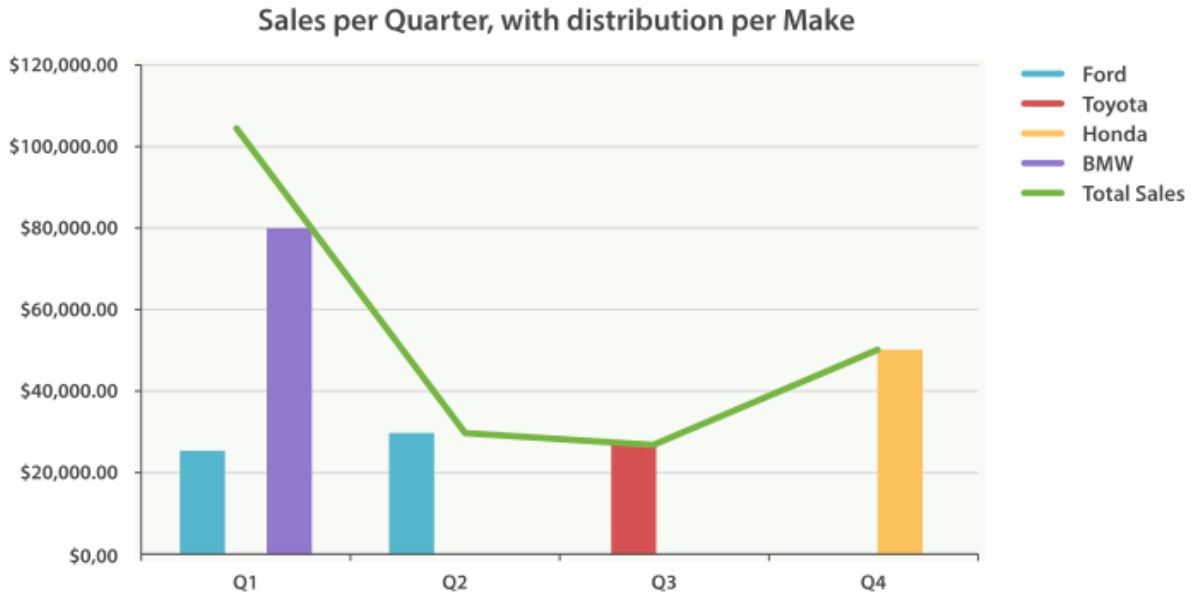

Figure 4. 按品牌分布的　季度

潜力无

　些很明　都是　　例子，但　表聚合其　是潜力无　的。 如 Kibana—用聚合　建的　　分析面板 展示了 Kibana 中用各　聚合　建的面板。

Figure 5. Kibana—用聚合　建的　　分析面板

因　聚合的　　性，　似　　的面板很容易　　、操作和交互。　使得它　成　需要分析数据又不会　建 Hadoop 作　的非技　人　的理想工具。

当然，　了　建　似　　Kibana　　的　大面板，　可能需要更深的知　，比如基于　　、

以及排序的聚合。

限定的聚合

所有聚合的例子到目前 止， 可能已 注意到，我 的搜索 求省略了一个 `query` 。 整个 求只不 是一个聚合。

聚合可以与搜索 求同 行，但是我 需要理解一个新概念： 。 情况下，聚合与 是同一 行操作的，也就是 ，聚合是基于我 匹配的文 集合 行 算的。

我 看看第一个聚合的示例：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "aggs" : {
        "colors" : {
            "terms" : {
              "field" : "color"
            }
        }
    }
}
```

我 可以看到聚合是隔 的。 中，Elasticsearch "没有指定 " 和 " 所有文 " 是等 的。前面 个 内部会 化成下面的 个 求：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "query" : {
        "match_all" : {}
    },
    "aggs" : {
        "colors" : {
            "terms" : {
              "field" : "color"
            }
        }
    }
}
```

因 聚合 是 内的 果 行操作的，所以一个隔 的聚合 上是在 `match_all` 的 果 操作，即所有的文 。

一旦有了 的概念，我 就能更 一 聚合 行自定 。我 前面所有的示例都是 所有 数据 算 信息的： 量最高的汽 ，所有汽 的平均 ，最佳 月 等等。

利用 ，我 可以 “福特在 有多少 色？” 如此 的 。可以 的在 求中加上一个 （本例中 `match` ）：

```
GET /cars/transactions/_search
{
    "query" : {
        "match" : {
            "make" : "ford"
        }
    },
    "aggs" : {
        "colors" : {
            "terms" : {
              "field" : "color"
            }
        }
    }
}
```

因 我 没有指定 `"size" : 0`，所以搜索 果和聚合 果都被返回了：

```
{
...
   "hits": {
      "total": 2,
      "max_score": 1.6931472,
      "hits": [
         {
            "_source": {
               "price": 25000,
               "color": "blue",
               "make": "ford",
               "sold": "2014-02-12"
            }
         },
         {
            "_source": {
               "price": 30000,
               "color": "green",
               "make": "ford",
               "sold": "2014-05-18"
            }
         }
      ]
   },
   "aggregations": {
      "colors": {
         "buckets": [
            {
               "key": "blue",
               "doc_count": 1
            },
            {
               "key": "green",
               "doc_count": 1
            }
         ]
      }
   }
}
```

看上去 并没有什 ，但却 高大上的 表 来 至 重要。 加入一个搜索 可以将任何静 的 表板 成一个 数据搜索 。 用 可以搜索数据， 看所有 更新的形（由于聚合的支持以及 的限定）。 是 Hadoop 无法做到的！

全局桶

通常我 希望聚合是在 内的，但有 我 也想要搜索它的子集，而聚合的 象却是 所有 数据。

例如，比方 我 想知道福特汽 与 所有 汽 平均 的比 。我 可以用普通的聚合（

内的）得到第一个信息，然后用 全局 桶 得第二个信息。

全局 桶包含 所有 的文 ，它无 的 。因 它 是一个桶，我 可以像平常一 将聚合嵌套在内：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "query" : {
        "match" : {
            "make" : "ford"
        }
    },
    "aggs" : {
        "single_avg_price": {
            "avg" : { "field" : "price" } ①
        },
        "all": {
            "global" : {}, ②
            "aggs" : {
                "avg_price": {
                    "avg" : { "field" : "price" } ③
                }

            }
        }
    }
}
```

① 聚合操作在 内（例如：所有文 匹配 ford ）

② global 全局桶没有参数。

③ 聚合操作 所有文 ，忽略汽 品牌。

single_avg_price 度量 算是基于 内所有文 ，即所有 福特 汽 。avg_price 度量是嵌套在 全局 桶下的， 意味着它完全忽略了 并 所有文 行 算。聚合返回的平均 是所有汽 的平均 。

如果能一直 持 到 里， 知道我 有个真言：尽可能的使用 器。它同 可以 用于聚合，在下一章中，我 会展示如何 聚合 果 行 而不是 做限定。

和聚合

聚合 限定 有一个自然的 展就是 。因 聚合是在 果 内操作的，任何可以 用于 的 器也可以 用在聚合上。

如果我 想 到 在 $10,000 美元之上的所有汽 同 也 些 算平均 ， 可以 地使用一个 constant_score 和 filter 束：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "query" : {
        "constant_score": {
            "filter": {
                "range": {
                    "price": {
                        "gte": 10000
                    }
                }
            }
        }
    },
    "aggs" : {
        "single_avg_price": {
            "avg" : { "field" : "price" }
        }
    }
}
```

正如我 在前面章 中 那 ，从根本上 ，使用 non-scoring 和使用 match 没有任何区 。 （包括了一个 器）返回一 文 的子集，聚合正是操作 些文 。使用 filtering query 会忽略 分，并有可能会 存 果数据等等。

桶

但是如果我 只想 聚合 果 ？ 假 我 正在 汽 商 建一个搜索 面， 我 希望 示用 搜索的 果，但是我 同 也想在 面上提供更 富的信息，包括（与搜索匹配的）上个月度汽 的平均 。

里我 无法 的做 限定，因 有 个不同的条件。搜索 果必 是 ford ，但是聚合 果必 足 ford AND sold > now - 1M 。

了解决 个 ，我 可以用一 特殊的桶，叫做 filter （注： 桶） 。 我 可以指定一个 桶，当文 足 桶的条件 ，我 将其加入到桶内。

果如下：

```
GET /cars/transactions/_search
{
   "size" : 0,
   "query":{
      "match": {
         "make": "ford"
      }
   },
   "aggs":{
      "recent_sales": {
         "filter": { ①
            "range": {
               "sold": {
                  "from": "now-1M"
               }
            }
         },
         "aggs": {
            "average_price":{
               "avg": {
                  "field": "price" ②
               }
            }
         }
      }
   }
}
```

① 使用　　桶在　　基　上　用　　器。

② avg 度量只会　ford 和上个月　出的文　　算平均　　。

因　　filter　桶和其他桶的操作方式一　，所以可以随意将其他桶和度量嵌入其中。所有嵌套的　件都会"　承"　个　　，　使我　可以按需　　聚合　　出　　部分。

后　　器

目前　止，我　可以同　　搜索　果和聚合　果　行　　（不　算得分的　filter　　），以及　　聚合　果的一部分　行　　（ filter 桶）。

我　可能会想，"只　　搜索　果，不　　聚合　果　？" 答案是使用 post_filter 。

它是接收一个　　器的　　搜索　求元素。　个　　器在　　之后　　行（　正是　　器的名字的由来：它在　　之后　　post　　行）。正因　它在　　之后　行，它没有任何影　，所以　聚合也不会有任何影　。

我　可以利用　个行　　条件　用更多的　　器，而不会影　其他的操作，就如　UI　上的各个分面。　我　　汽　　商　　外一个搜索　面，　个　面允　用　搜索汽　同　可以根据　色来　　。色的　　是通　聚合　得的：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "query": {
        "match": {
            "make": "ford"
        }
    },
    "post_filter": {    ①
        "term" : {
            "color" : "green"
        }
    },
    "aggs" : {
        "all_colors": {
            "terms" : { "field" : "color" }
        }
    }
}
```

① post_filter 元素是 top-level 而且 命中 果 行 。

部分 到所有的 ford 汽 ，然后用 terms 聚合 建一个 色列表。因 聚合 行操作，色列表与福特汽 有的 色相 。

最后， post_filter 会 搜索 果，只展示 色 ford 汽 。 在 行 后 生，所以聚合不受影 。

通常 UI 的 一致性很重要，可以想象用 在界面商 了一 色（比如：色），期望的是搜索 果已 被 了，而 不是 界面上的 。如果我 用 filter ，界面会上 成只 示 色作 ， 不是用 想要的！

WARNING

性能考 （*Performance consideration*）

当 需要 搜索 果和聚合 果做不同的 ， 才 使用 post_filter ， 有用 会在普通搜索使用 post_filter 。

不要 做！ post_filter 的特性是在 之后 行，任何 性能 来的好（比如 存）都会完全失去。

在我 需要不同 ， post_filter 只与聚合一起使用。

小

合 型的 （如：搜索命中、聚合或 者兼有）通常和我 期望如何表 用 交互有 。 合的 器（或 合）取决于我 期望如何将 果呈 用 。

- 在 filter 中的 non-scoring ，同 影 搜索 果和聚合 果。
- filter 桶影 聚合。

- post_filter 只影 搜索 果。

多桶排序

多 桶（ terms 、 histogram 和 date_histogram ） 生成很多桶。 Elasticsearch 是如何决定 些桶展示 用 的 序 ？

的，桶会根据 doc_count 降序排列。 是一个好的 行 ，因 通常我 想要 到文 中与 条件相 的最大 ： 、人口数量、 率。但有些 候我 希望能修改 个 序，不同的桶有着不同的 理方式。

内置排序

些排序模式是桶 固有的 能力：它 操作桶生成的数据 ，比如 doc_count 。 它 共享相同的 法，但是根据使用桶的不同会有些 微差 。

我 做一个 terms 聚合但是按 doc_count 的升序排序：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "aggs" : {
        "colors" : {
            "terms" : {
              "field" : "color",
              "order": {
                "_count" : "asc" ①
              }
            }
        }
    }
}
```

① 用 字 _count ，我 可以按 doc_count 的升序排序。

我 聚合引入了一个 order 象， 它允 我 可以根据以下几个 中的一个 行排序：

_count
按文 数排序。 terms 、 histogram 、 date_histogram 有效。

_term
按 的字符串 的字母 序排序。只在 terms 内使用。

_key
按 个桶的 数 排序（理 上与 _term 似）。 只在 histogram 和 date_histogram 内使用。

按度量排序

有 ，我 会想基于度量 算的 果 行排序。 在我 的汽 分析 表 中，我 可能想按照汽 色 建一个 条状 表，但按照汽 平均 的升序 行排序。

我　可以　加一个度量，再指定 order 参数引用　个度量即可：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "aggs" : {
        "colors" : {
            "terms" : {
              "field" : "color",
              "order": {
                "avg_price" : "asc" ②
              }
            },
            "aggs": {
                "avg_price": {
                    "avg": {"field": "price"} ①
                }
            }
        }
    }
}
```

①　算　个桶的平均　　。

② 桶按照　算平均　的升序排序。

我　可以采用　　方式用任何度量排序，只需　　的引用度量的名字。不　有些度量会　出多个　。`extended_stats` 度量是一个很好的例子：它　出好几个度量　。

如果我　想使用多　度量　行排序，　我　只需以　心的度量　　使用点式路径：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "aggs" : {
        "colors" : {
            "terms" : {
              "field" : "color",
              "order": {
                "stats.variance" : "asc" ①
              }
            },
            "aggs": {
                "stats": {
                    "extended_stats": {"field": "price"}
                }
            }
        }
    }
}
```

① 使用 . 符号，根据感 趣的度量 行排序。

在上面 个例子中，我 按 个桶的方差来排序，所以 色 方差最小的会排在 果集最前面。

基于“深度”度量排序

在前面的示例中，度量是桶的直接子 点。平均 是根据 个 term 来 算的。 在一定条件下，我 也有可能 更深 的度量 行排序，比如 子桶或从 桶。

我 可以定 更深的路径，将度量用尖括号（ > ）嵌套起来，像 ： my_bucket>another_bucket>metric 。

需要提醒的是嵌套路径上的 个桶都必 是 的。 filter 桶生成 一个 桶：所有与条件匹配的文 都在桶中。多 桶（如：terms ） 生成 多桶，无法通 指定一个 定路径来 。

目前，只有三个 桶： filter 、 global 和 reverse_nested 。 我 快速用示例 明，建一个汽 的直方 ，但是按照 色和 色（不包括 色） 各自的方差来排序：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "aggs" : {
        "colors" : {
            "histogram" : {
              "field" : "price",
              "interval": 20000,
              "order": {
                "red_green_cars>stats.variance" : "asc" ①
              }
            },
            "aggs": {
                "red_green_cars": {
                    "filter": { "terms": {"color": ["red", "green"]}}, ②
                    "aggs": {
                        "stats": {"extended_stats": {"field" : "price"}} ③
                    }
                }
            }
        }
    }
}
```

① 按照嵌套度量的方差 桶的直方 行排序。

② 因 我 使用 器 `filter`，我 可以使用嵌套排序。

③ 按照生成的度量 果 行排序。

本例中，可以看到我 如何 一个嵌套的度量。 `stats` 度量是 `red_green_cars` 聚合的子 点，而 `red_green_cars` 又是 `colors` 聚合的子 点。 了根据 个度量排序，我 定 了路径 `red_green_cars>stats.variance`。我 可以 做，因 `filter` 桶是个 桶。

近似聚合

如果所有的数据都在一台机器上，那 生活会容易 多。 CS201 上教的 典算法就足 付 些 。如果所有的数据都在一台机器上，那 也就不需要像 Elasticsearch 的分布式 件了。不一旦我 始分布式存 数据，就需要小心地 算法。

有些算法可以分布 行，到目前 止 的所有聚合都是 次 求 得精 果的。 些 型的算法通常被 是 高度并行的 ，因 它 无 任何 外代 ，就能在多台机器上并行 行。比如当 算 `max` 度量 ，以下的算法就非常 ：

1. 把 求广播到所有分片。
2. 看 个文 的 price 字段。如果 `price > current_max`，将 `current_max` 替 成 `price`。
3. 返回所有分片的最大 price 并 点。
4. 到从所有分片返回的最大 price。 是最 的最大 。

个算法可以随着机器数的 性 而横向 展，无 任何 操作（机器之 不需要 中 果），而且内存消耗很小（一个整型就能代表最大 ）。

不幸的是，不是所有的算法都像 取最大 。更加 的操作 需要在算法的性能和内存使用上做出 衡。 于 个 ，我 有个三角因子模型：大数据、精 性和 性。

我 需要 其中 ：

精 +

数据可以存入 台机器的内存之中，我 可以随心所欲，使用任何想用的算法。 果会 100% 精 ， 会相 快速。

大数据 + 精

的 Hadoop。可以 理 PB 的数据并且 我 提供精 的答案，但它可能需要几周的 才能 我 提供 个答案。

大数据 +

近似算法 我 提供准 但不精 的 果。

Elasticsearch 目前支持 近似算法（ `cardinality` 和 `percentiles` ）。 它 会提供准 但不是 100% 精 的 果。 以 牲一点小小的估算 代 ， 些算法可以 我 来高速的 行效率和小的内存消耗。

于 大多数 用 域，能 返回高度准 的 果要比 100% 精 果重要得多。乍一看可能是天方夜 。有人会叫 “我 需要精 的答案！”。但仔 考 0.5% 差所 来的影 ：

- 99% 的 站延 都在 132ms 以下。
- 0.5% 的 差 以上延 的影 在正 0.66ms 。
- 近似 算的 果会在 秒内返回，而“完全正 ”的 果就可能需要几秒，甚至无法返回。

只要 的 看 站的延 情况， 道我 会在意近似 果是 132.66ms 而不是 132ms ？当然，不是所有的 域都能容忍 近似 果，但 于 大多数来 是没有 的。接受近似 果更多的是一 文化 念上 的壁 而不是商 或技 上的需要。

去重后的数量

Elasticsearch 提供的首个近似聚合是 `cardinality` （注：基数）度量。 它提供一个字段的基数，即字段的 *distinct* 或者 *unique* 的数目。 可能会 SQL 形式比 熟悉：

```
SELECT COUNT(DISTINCT color)
FROM cars
```

去重是一个很常 的操作，可以回答很多基本的 ：

- 站独立 客是多少？
- 了多少 汽 ？
- 月有多少独立用 了商品？

我 可以用 cardinality 度量 定 商 汽 色的数量：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "aggs" : {
        "distinct_colors" : {
            "cardinality" : {
              "field" : "color"
            }
        }
    }
}
```

返回的 果表明已 了三 不同 色的汽 ：

```
...
"aggregations": {
  "distinct_colors": {
     "value": 3
  }
}
...
```

可以 我 的例子 得更有用： 月有多少 色的 被 出？ 了得到 个度量，我 只需要将一个 cardinality 度量嵌入一个 date_histogram ：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
      "months" : {
        "date_histogram": {
          "field": "sold",
          "interval": "month"
        },
        "aggs": {
          "distinct_colors" : {
              "cardinality" : {
                "field" : "color"
              }
          }
        }
      }
  }
}
```

学会 衡

正如我 本章 提到的， cardinality 度量是一个近似算法。 它是基于 HyperLogLog++ （HLL）算法的。 HLL 会先 我 的 入作哈希 算，然后根据哈希 算的 果中的 bits 做概率估算从而得到基数。

我 不需要理解技 （如果 感 趣，可以 篇 文）， 但我 最好 注一下 个算法的特性 ：

- 可配置的精度，用来控制内存的使用（更精 = 更多内存）。
- 小的数据集精度是非常高的。
- 我 可以通 配置参数，来 置去重需要的固定内存使用量。无 数千 是数十 的唯一 ，内存使用量只与 配置的精 度相 。

要配置精度，我 必 指定 precision_threshold 参数的 。 个 定 了在何 基数水平下我希望得到一个近乎精 的 果。参考以下示例：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "aggs" : {
        "distinct_colors" : {
            "cardinality" : {
              "field" : "color",
              "precision_threshold" : 100 ①
            }
        }
    }
}
```

① precision_threshold 接受 0–40,000 之 的数字，更大的 是会被当作 40,000 来 理。

示例会 保当字段唯一 在 100 以内 会得到非常准 的 果。尽管算法是无法保点的，但如果基数在 以下，几乎 是 100% 正 的。高于 的基数会 始 省内存而 牲准度，同 也会 度量 果 入 差。

于指定的 ，HLL 的数据 会大概使用 precision_threshold * 8 字 的内存，所以就必 在牲内存和 得 外的准 度 做平衡。

在 用中， 100 的 可以在唯一 百万的情况下 然将 差 持 5% 以内。

速度 化

如果想要 得唯一 的数目， 通常 需要 整个数据集合（或几乎所有数据）。所有基于所有数据的操作都必 迅速，原因是 然的。 HyperLogLog 的速度已 很快了，它只是 的数据做哈希以及一些位操作。

但如果速度 我 至 重要，可以做 一 的 化。 因 HLL 只需要字段内容的哈希 ，我可以在索引 就 先 算好。就能在 跳 哈希 算然后将哈希 从 fielddata 直接加 出来。

先 算哈希 只 内容很 或者基数很高的字段有用， 算 些字段的哈希 的消耗在 是无法忽略的。

NOTE

尽管数 字段的哈希 算是非常快速的，存 它 的原始 通常需要同 （或更少）的内存空 。 低基数的字符串字段同 用，Elasticsearch 的内部 化能 保 个唯一 只 算一次哈希。

基本上 ， 先 算并不能保 所有的字段都更快，它只 那些具有高基数和/或者内容很 的字符串字段有作用。需要 住的是， 算只是 的将 消耗的 提前 移到索引 ，并非没有任何代 ，区 在于 可以 在 什 候 做 件事，要 在索引 ，要 在 。

要想 做，我 需要 数据 加一个新的多 字段。我 先 除索引，再 加一个包括哈希 字段的映射 ，然后重新索引：

```
DELETE /cars/

PUT /cars/
{
  "mappings": {
    "transactions": {
      "properties": {
        "color": {
          "type": "string",
          "fields": {
            "hash": {
              "type": "murmur3" ①
            }
          }
        }
      }
    }
  }
}

POST /cars/transactions/_bulk
{ "index": {}}
{ "price" : 10000, "color" : "red", "make" : "honda", "sold" : "2014-10-28" }
{ "index": {}}
{ "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05" }
{ "index": {}}
{ "price" : 30000, "color" : "green", "make" : "ford", "sold" : "2014-05-18" }
{ "index": {}}
{ "price" : 15000, "color" : "blue", "make" : "toyota", "sold" : "2014-07-02" }
{ "index": {}}
{ "price" : 12000, "color" : "green", "make" : "toyota", "sold" : "2014-08-19" }
{ "index": {}}
{ "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05" }
{ "index": {}}
{ "price" : 80000, "color" : "red", "make" : "bmw", "sold" : "2014-01-01" }
{ "index": {}}
{ "price" : 25000, "color" : "blue", "make" : "ford", "sold" : "2014-02-12" }
```

① 多 字段的 型是 murmur3 ， 是一个哈希函数。

在当我 行聚合 ，我 使用 color.hash 字段而不是 color 字段：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "aggs" : {
        "distinct_colors" : {
            "cardinality" : {
              "field" : "color.hash" ①
            }
        }
    }
}
```

① 注意我 指定的是哈希 的多 字段，而不是原始字段。

在 cardinality 度量会 取 "color.hash" 里的 （ 先 算的哈希 ），取代 算原始 的哈希。

个文 省的 是非常少的，但是如果 聚合一 数据， 个字段多花 10 秒的 ，那 在次 都会 外 加 1 秒，如果我 要在非常大量的数据里面使用 cardinality ，我 可以 衡使用 算的意 ，是否需要提前 算 hash，从而在 得更好的性能，做一些性能 来 算哈希是否 用于 的 用 景。。

百分位 算

Elasticsearch 提供的 外一个近似度量就是 percentiles 百分位数度量。 百分位数展某以具体百分比下 察到的数 。例如，第95个百分位上的数 ，是高于 95% 的数据 和。

百分位数通常用来 出 常。在（ 学）的正 分布下，第 0.13 和 第 99.87 的百分位数代表与均 距 三倍 准差的 。任何 于三倍 准差之外的数据通常被 是不 常的，因 它与平均 相差太大。

更具体的 ，假 我 正 行一个 大的 站，一个很重要的工作是保 用 求能得到快速 ，因此我 就需要 控 站的延 来判断 是否能保 良好的用 体 。

在此 景下，一个常用的度量方法就是平均 延 。 但 并不是一个好的 （尽管很常用），因 平均数通常会 藏那些 常 ， 中位数有着同 的 。 我 可以 最大 ，但 个度量会 而易 的被 个 常 破坏。

在 Average request latency over time 看 。如果我 依 如平均 或中位数 的 度量，就会得到像 一幅 Average request latency over time 。

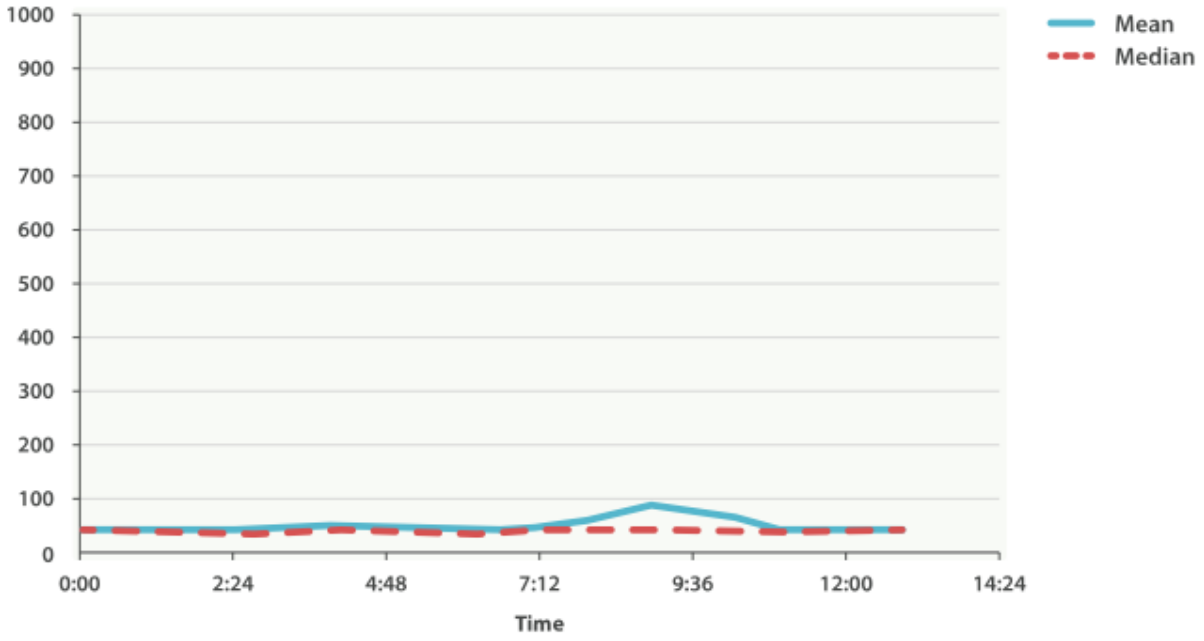

Figure 6. Average request latency over time

一切正常。 上有 微的波 ，但没有什 得 注的。 但如果我 加 99 百分位数 （ 个 代表最慢的 1% 的延 ），我 看到了完全不同的一幅画面，如 Average request latency with 99th percentile over time 。

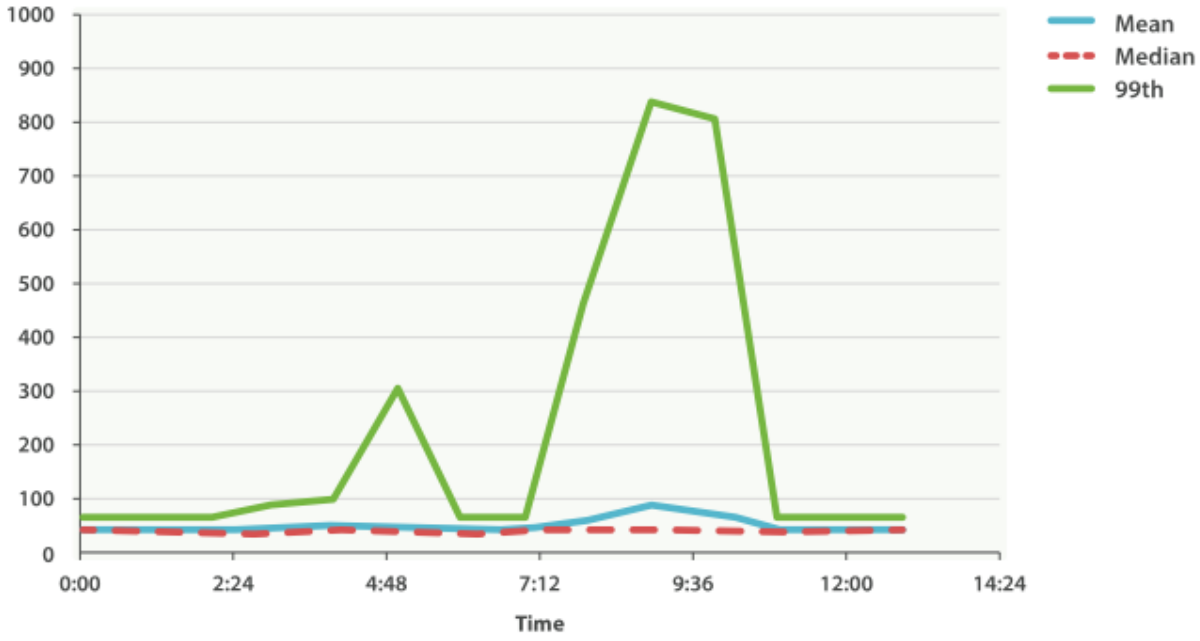

Figure 7. Average request latency with 99th percentile over time

令人吃 ！在上午九点半 ，均 只有 75ms。如果作 一个系 管理 ，我 都不会看他第二眼。一切正常！但 99 百分位告 我 有 1% 的用 到的延 超 850ms， 是 外一幅 景。 在上午4点48 也有一个小波 ， 甚至无法从平均 和中位数曲 上 察到。

只是百分位的一个 用 景，百分位 可以被用来快速用肉眼 察数据的分布， 是否有数据 斜或双峰甚至更多。

百分位度量

我加 一个新的数据集（汽 的数据不太 用于百分位）。我 要索引一系列 站延 数据然后 行一些百分位操作 行 看：

```
POST /website/logs/_bulk
{ "index": {}}
{ "latency" : 100, "zone" : "US", "timestamp" : "2014-10-28" }
{ "index": {}}
{ "latency" : 80, "zone" : "US", "timestamp" : "2014-10-29" }
{ "index": {}}
{ "latency" : 99, "zone" : "US", "timestamp" : "2014-10-29" }
{ "index": {}}
{ "latency" : 102, "zone" : "US", "timestamp" : "2014-10-28" }
{ "index": {}}
{ "latency" : 75, "zone" : "US", "timestamp" : "2014-10-28" }
{ "index": {}}
{ "latency" : 82, "zone" : "US", "timestamp" : "2014-10-29" }
{ "index": {}}
{ "latency" : 100, "zone" : "EU", "timestamp" : "2014-10-28" }
{ "index": {}}
{ "latency" : 280, "zone" : "EU", "timestamp" : "2014-10-29" }
{ "index": {}}
{ "latency" : 155, "zone" : "EU", "timestamp" : "2014-10-29" }
{ "index": {}}
{ "latency" : 623, "zone" : "EU", "timestamp" : "2014-10-28" }
{ "index": {}}
{ "latency" : 380, "zone" : "EU", "timestamp" : "2014-10-28" }
{ "index": {}}
{ "latency" : 319, "zone" : "EU", "timestamp" : "2014-10-29" }
```

数据有三个 ：延 、数据中心的区域以及 。 我 数据全集 行 百分位 操作以 得数据分布情况的直 感受：

```
GET /website/logs/_search
{
    "size" : 0,
    "aggs" : {
        "load_times" : {
            "percentiles" : {
                "field" : "latency" ①
            }
        },
        "avg_load_time" : {
            "avg" : {
                "field" : "latency" ②
            }
        }
    }
}
```

① percentiles 度量被　用到 latency 延　字段。

②　了比　，我　　相同字段使用 avg 度量。

　情况下，percentiles 度量会返回一　　定　的百分位数　：[1, 5, 25, 50, 75, 95, 99] 。它表示了人　感　趣的常用百分位数　，　端的百分位数在　　的　　，其他的一些　于中部。在返回的　　中，我　可以看到最小延　在 75ms 左右，而最大延　差不多有 600ms。与之形成　比的是，平均延　在 200ms 左右，信息并不是很多：

```
...
"aggregations": {
  "load_times": {
     "values": {
        "1.0": 75.55,
        "5.0": 77.75,
        "25.0": 94.75,
        "50.0": 101,
        "75.0": 289.75,
        "95.0": 489.34999999999985,
        "99.0": 596.2700000000002
     }
  },
  "avg_load_time": {
     "value": 199.58333333333334
  }
}
```

所以　然延　的分布很广，　我　看看它　是否与数据中心的地理区域有　：

```
GET /website/logs/_search
{
    "size" : 0,
    "aggs" : {
        "zones" : {
            "terms" : {
                "field" : "zone" ①
            },
            "aggs" : {
                "load_times" : {
                    "percentiles" : { ②
                      "field" : "latency",
                      "percents" : [50, 95.0, 99.0] ③
                    }
                },
                "load_avg" : {
                    "avg" : {
                        "field" : "latency"
                    }
                }
            }
        }
    }
}
```

① 首先根据区域我　将延　分到不同的桶中。

② 再　算　个区域的百分位数　。

③ percents 参数接受了我　想返回的一　百分位数，因　我　只　　的延　感　趣。

在　　果中，我　　欧洲区域（EU）要比美国区域（US）慢很多，在美国区域（US），50　百分位与 99 百分位十分接近，它　都接近均　。

与之形成　比的是，欧洲区域（EU）在 50 和 99 百分位有　大区分。　在，　然可以　　是欧洲区域（EU）拉低了延　的　　信息，我　知道欧洲区域的 50% 延　都在 300ms+。

```
...
"aggregations": {
  "zones": {
     "buckets": [
        {
           "key": "eu",
           "doc_count": 6,
           "load_times": {
              "values": {
                 "50.0": 299.5,
                 "95.0": 562.25,
                 "99.0": 610.85
              }
           },
           "load_avg": {
              "value": 309.5
           }
        },
        {
           "key": "us",
           "doc_count": 6,
           "load_times": {
              "values": {
                 "50.0": 90.5,
                 "95.0": 101.5,
                 "99.0": 101.9
              }
           },
           "load_avg": {
              "value": 89.66666666666667
           }
        }
     ]
  }
}
...
```

百分位等

里有 外一个 密相 的度量叫 percentile_ranks 。 percentiles 度量告 我 落在某个百分比以下的所有文 的最小 。例如，如果 50 百分位是 119ms，那 有 50% 的文 数 都不超 119ms。percentile_ranks 告 我 某个具体 属于 个百分位。119ms 的 percentile_ranks 是在 50 百分位。 基本是个双向 系，例如：

- 50 百分位是 119ms。
- 119ms 百分位等 是 50 百分位。

所以假 我 站必 持的服 等 （SLA）是 低于 210ms。然后， 个玩笑，我 老板警告我 如果 超 800ms 会把我 除。可以理解的是，我 希望知道有多少百分比的

求可以　足 SLA 的要求（并期望至少在 800ms 以下！）。

了做到　点，我　可以　用 percentile_ranks 度量而不是 percentiles 度量：

```
GET /website/logs/_search
{
    "size" : 0,
    "aggs" : {
        "zones" : {
            "terms" : {
                "field" : "zone"
            },
            "aggs" : {
                "load_times" : {
                    "percentile_ranks" : {
                      "field" : "latency",
                      "values" : [210, 800] ①
                    }
                }
            }
        }
    }
}
```

① percentile_ranks 度量接受一　我　希望分　的数　。

在聚合　行后，我　能得到　个　：

```
"aggregations": {
  "zones": {
     "buckets": [
        {
           "key": "eu",
           "doc_count": 6,
           "load_times": {
              "values": {
                 "210.0": 31.944444444444443,
                 "800.0": 100
              }
           }
        },
        {
           "key": "us",
           "doc_count": 6,
           "load_times": {
              "values": {
                 "210.0": 100,
                 "800.0": 100
              }
           }
        }
     ]
  }
}
```

告 我 三点重要的信息：

- 在欧洲（EU），210ms 的百分位等 是 31.94% 。
- 在美国（US），210ms 的百分位等 是 100% 。
- 在欧洲（EU）和美国（US），800ms 的百分位等 是 100% 。

通俗的 ，在欧洲区域（EU）只有 32% 的 足服 等 （SLA），而美国区域（US）始 足服 等 的。但幸 的是， 个区域所有 都在 800ms 以下，所以我 不会被炒 （至少目前不会）。

`percentile_ranks` 度量提供了与 `percentiles` 相同的信息，但它以不同方式呈 ，如果我 某个具体数 更 心，使用它会更方便。

学会 衡

和基数一 ， 算百分位需要一个近似算法。 朴素的 会 一个所有 的有序列表， 但当我 有几十 数据分布在几十个 点 ， 几乎是不可能的。

取而代之的是 `percentiles` 使用一个 TDigest 算法，（由 Ted Dunning 在 Computing Extremely Accurate Quantiles Using T-Digests 里面提出的）。 与 HyperLogLog 一 ，不需要理解完整的技 ，但有必要了解算法的特性：

- 百分位的准　度与百分位的　　端程度　相　，也就是　　1　或　99　的百分位要比　50　百分位要准　。　只是数据　　内部机制的一　特性，但　是一个好的特性，因　多数人只　心　端的百分位。
- 　于数　集合　小的情况，百分位非常准　。如果数据集足　小，百分位可能 100% 精　。
- 随着桶里数　的　　，算法会　始　百分位　行估算。它能有效在准　度和内存　省之　做出　衡。　不准　的程度比　　以　　，因　它依　于 聚合　数据的分布以及数据量的大小。

与 `cardinality`　似，我　可以通　修改参数 `compression` 来控制内存与准　度之　的比　。

TDigest　　算法用　点近似　算百分比：　点越多，准　度越高（同　内存消耗也越大），　都与数据量成正比。`compression` 参数限制　点的最大数目　`20 * compression`。

因此，通　　加　　比　，可以以消耗更多内存　代　提高百分位数准　性。更大的　　比　会使算法　行更慢，因　底　的　形数据　　的存　也会　　，也　致操作的代　更高。　　的　　比　是 `100`。

一个　点大　使用　　32　　字　的内存，所以在最坏的情况下（例如，大量数据有序存入），　　置会生成一个大小　　64KB　的　TDigest。　在　　用中，数据会更随机，所以　TDigest 使用的内存会更少。

通　聚合　　常指

`significant_terms`　（SigTerms）聚合　与其他聚合都不相同。　目前　止我　看到的所有聚合在本上都是　　的数学　算。将不同　些　造　相互　合在一起，我　可以　建　　的聚合以及数据　表。

`significant_terms`　　有着不同的工作机制。　有些人来　，它甚至看起来有点像机器学　。`significant_terms` 聚合可以在　数据集中　到一些　常 的指　。

如何解　　些　不常　　的行　？　　些　常的数据指　通常比我　　估出　的　次要更　繁，　些　　上的　常指　通常象征着数据里的某些有趣信息。

例如，假　我　　和跟踪信用　欺　，客　打　　来抱怨他　信用　出　　常交易，它　的　　已　被盗用。　些交易信息只是更　重　　的症状。在最近的某些地区，一些商家有意的盗取客　的信用　信息，或者它　自己的信息无意中也被盗取。

我　的任　是　到　危害的共同点　，如果我　有　100　个客　抱怨交易　常，他　很有可能都属于同一个商　，而　家商　有可能就是罪魁　首。

当然，　里面　有一些特例。例如，很多客　在它　近期交易　史　　中会有很大的商　如　　，我　可以将　　排除在外，然而，在最近一些有　　的信用　的商家里面也有　　。

　是一个 普通的共同 商　的例子。　个人都共享　个商　，无　有没有遭受危害。我　　它并不感　趣。

相反，我　有一些很小商　比如街角的一家　店，它　属于　　普通但不　常　　的情况，只有一　个客有交易　　。我　同　可以将　些商　排除，因　所有受到危害的信用　都没有与　些商　　生　交易，我　可以肯定它　不是安全漏洞的　任方。

我　真正想要的是　　不普通的共同　　商　。所有受到危害的信用　都与它　　生　交易，但是在未受危害的背景噪声下，它　并不明　。　些商　属于　　常，它　比　　出　的　率要高。　些不普通的共同商　很有可能就是需要　　的。

`significant_terms`　　聚合就是做　些事情。它分析　　的数据并通　　比正常数据　到可能有　常

次的指 。

暴露的 常指 代表什 依 的 数据。 于信用 数据，我 可能会想 出信用 欺 。 于 商数据，我 可能会想 出未被 的人口信息，从而 行更高效的市 推广。 如果我 正在分析日志，我 可能会 一个服 器会 出比它本 出的更多 常。 `significant_terms` 的用 不止 些。

significant_terms 演示

因 `significant_terms` 聚合 是通 分析 信息来工作的， 需要 数据 置一个 它 更有效。也就是 无法通 只索引少量示例数据来展示它。

正因如此，我 准 了一个大 8000 个文 的数据集，并将它的快照保存在一个公共演示 中。可以通 以下 在集群中 原 些数据：

1. 在 `elasticsearch.yml` 配置文件中 加以下配置， 以便将演示 加入到白名 中：

```
repositories.url.allowed_urls: ["http://download.elastic.co/*"]
```

2. 重 Elasticsearch。
3. 行以下快照命令。（更多使用快照的信息，参 集群（Backing Up Your Cluster））。

```
PUT /_snapshot/sigterms ①
{
    "type": "url",
    "settings": {
        "url": "http://download.elastic.co/definitiveguide/sigterms_demo/"
    }
}

GET /_snapshot/sigterms/_all ②

POST /_snapshot/sigterms/snapshot/_restore ③

GET /mlmovies,mlratings/_recovery ④
```

① 注 一个新的只 地址 ，并指向演示快照。

② （可 ） 内 于快照的 信息。

③ 始 原 程。会在集群中 建 个索引： `mlmovies` 和 `mlratings` 。

④ （可 ）使用 Recovery API 控 原 程。

NOTE 数据集有 50 MB 会需要一些 下 。

在本演示中，会看看 MovieLens 里面用 影的 分。在 MovieLens 里，用 可以推 影并分， 其他用 也可以 到新的 影。 了演示，会基于 入的 影采用 `significant_terms` 影 行推 。

我 看看示例中的数据，感受一下要 理的内容。本数据集有 个索引， mlmovies 和 mlratings 。首先 看 mlmovies ：

```
GET mlmovies/_search ①

{
   "took": 4,
   "timed_out": false,
   "_shards": {...},
   "hits": {
      "total": 10681,
      "max_score": 1,
      "hits": [
         {
            "_index": "mlmovies",
            "_type": "mlmovie",
            "_id": "2",
            "_score": 1,
            "_source": {
               "offset": 2,
               "bytes": 34,
               "title": "Jumanji (1995)"
            }
         },
         ....
```

① 行一个不 条件的搜索，以便能看到一 随机演示文 。

mlmovies 里的 个文 表示一个 影，数据有 个重要字段： 影ID _id 和 影名 title 。可以忽略 offset 和 bytes 。它 是从原始 CSV 文件抽取数据的 程中 生的中 属性。数据集中有 10，681 部影片。

在来看看 mlratings ：

```
GET mlratings/_search

{
   "took": 3,
   "timed_out": false,
   "_shards": {...},
   "hits": {
      "total": 69796,
      "max_score": 1,
      "hits": [
         {
            "_index": "mlratings",
            "_type": "mlrating",
            "_id": "00IC-2jDQFiQkpD6vhbFYA",
            "_score": 1,
            "_source": {
               "offset": 1,
               "bytes": 108,
               "movie": [122,185,231,292,
                  316,329,355,356,362,364,370,377,420,
                  466,480,520,539,586,588,589,594,616
               ],
               "user": 1
            }
         },
         ...
```

里可以看到 个用 的推 信息。 个文 表示一个用 ，用 ID 字段 `user` 来表示， `movie` 字段 一个用 看和推 的影片列表。

基于流行程度推 （Recommending Based on Popularity）

可以采取的首个策略就是基于流行程度向用 推 影片。 于某部影片， 到所有推 它的用 ，然后将他 的推 行聚合并 得推 中最流行的五部。

我 可以很容易的通 一个 `terms` 聚合 以及一些 来表示它，看看 *Talladega Nights*（塔拉 加之夜） 部影片，它是 Will Ferrell 主演的一部 于全国 汽 （NASCAR racing）的喜 。在理想情况下，我 的推 到 似 格的喜 （很有可能也是 Will Ferrell 主演的）。

首先需要 到影片 *Talladega Nights* 的 ID：

```
GET mlmovies/_search
{
  "query": {
    "match": {
      "title": "Talladega Nights"
    }
  }
}


    ...
    "hits": [
     {
        "_index": "mlmovies",
        "_type": "mlmovie",
        "_id": "46970", ①
        "_score": 3.658795,
        "_source": {
           "offset": 9575,
           "bytes": 74,
           "title": "Talladega Nights: The Ballad of Ricky Bobby (2006)"
        }
     },
    ...
```

① *Talladega Nights* 的 ID 是 `46970`。

有了 ID，可以　　分，再　用 `terms` 聚合从喜　*Talladega Nights* 的用　中　到最流行的影片：

```
GET mlratings/_search
{
  "size" : 0, ①
  "query": {
    "filtered": {
      "filter": {
        "term": {
          "movie": 46970 ②
        }
      }
    }
  },
  "aggs": {
    "most_popular": {
      "terms": {
        "field": "movie", ③
        "size": 6
      }
    }
  }
}
```

① 次 mlratings，将 果内容大小 置 0 因 我 只 聚合的 果感 趣。

② 影片 *Talladega Nights* 的 ID 使用 器。

③ 最后，使用 terms 桶 到最流行的影片。

在 mlratings 索引下搜索，然后 影片 *Talladega Nights* 的 ID 使用 器。由于聚合是 行操作的，它可以有效的 聚合 果从而得到那些只推 *Talladega Nights* 的用 。最后， 行 terms 聚合得到最流行的影片。 求排名最前的六个 果，因 *Talladega Nights* 本身很有可能就是其中一个 果（并不想重 推 它）。

返回 果就像 ：

```
{
...
   "aggregations": {
      "most_popular": {
         "buckets": [
            {
               "key": 46970,
               "key_as_string": "46970",
               "doc_count": 271
            },
            {
               "key": 2571,
               "key_as_string": "2571",
               "doc_count": 197
            },
            {
               "key": 318,
               "key_as_string": "318",
               "doc_count": 196
            },
            {
               "key": 296,
               "key_as_string": "296",
               "doc_count": 183
            },
            {
               "key": 2959,
               "key_as_string": "2959",
               "doc_count": 183
            },
            {
               "key": 260,
               "key_as_string": "260",
               "doc_count": 90
            }
         ]
      }
   }
...
```

通 一个 的 ，将得到的 果 成原始影片名：

```
GET mlmovies/_search
{
  "query": {
    "filtered": {
      "filter": {
        "ids": {
          "values": [2571,318,296,2959,260]
        }
      }
    }
  }
}
```

最后得到以下列表：

1. Matrix, The（ 客帝国）
2. Shawshank Redemption（肖申克的救 ）
3. Pulp Fiction（低俗小 ）
4. Fight Club（搏 部）
5. Star Wars Episode IV: A New Hope（星球大 IV：曙光乍 ）

好 ， 肯定不是一个好的列表！我喜 所有 些影片。但 是：几乎 个人 都喜 它 。 些影片本来就受大 迎，也就是 它 出 在 个人的推 中都会受 迎。 其是一个流行影片的推 列表，而不是和影片 *Talladega Nights* 相 的推 。

可以通 再次 行聚合 松 ，而不需要 影片 *Talladega Nights* 行 。会提供最流行影片的前五名列表：

```
GET mlratings/_search
{
  "size" : 0,
  "aggs": {
    "most_popular": {
      "terms": {
        "field": "movie",
        "size": 5
      }
    }
  }
}
```

返回列表非常相似：

1. Shawshank Redemption（肖申克的救 ）
2. Silence of the Lambs, The（ 的 羊）
3. Pulp Fiction（低俗小 ）

4. Forrest Gump（阿甘正　）
5. Star Wars Episode IV: A New Hope（星球大　IV：曙光乍　）

然，只是　　最流行的影片是不能足以　建一个良好而又具　　能力的推　系　。

基于　　的推　（Recommending Based on Statistics）

在　景已　　定好，使用 significant_terms 。 significant_terms 会分析喜　影片 *Talladega Nights* 的用　　（　　前端　　用　　），并且　定最流行的　影。　　然后　　个用　（　　后端　　用　）　造一个流行影片列表，最后将　者　行比　。

　　常就是与　　背景相比在前景特征　中　度展　的那些影片。理　上　，它　　是一　喜　，因　喜　Will Ferrell 喜　的人　　些影片的　分会比一般人高。

我　　一下：

```
GET mlratings/_search
{
  "size" : 0,
  "query": {
    "filtered": {
      "filter": {
        "term": {
          "movie": 46970
        }
      }
    }
  },
  "aggs": {
    "most_sig": {
      "significant_terms": { ①
        "field": "movie",
        "size": 6
      }
    }
  }
}
```

① 　置几乎一模一　，只是用 significant_terms 替代了 terms 。

正如所　，　　也几乎是一　的。　　出喜　影片 *Talladega Nights* 的用　，他　　成了前景特征用　。　　情况下， significant_terms 会使用整个索引里的数据作　　背景，所以不需要特　的　理。

与 terms　似，　果返回了一　桶，不　有更多的元数据信息：

```
...
   "aggregations": {
      "most_sig": {
         "doc_count": 271, ①
         "buckets": [
            {
               "key": 46970,
               "key_as_string": "46970",
               "doc_count": 271,
               "score": 256.549815498155,
               "bg_count": 271
            },
            {
               "key": 52245, ②
               "key_as_string": "52245",
               "doc_count": 59, ③
               "score": 17.66462367106966,
               "bg_count": 185 ④
            },
            {
               "key": 8641,
               "key_as_string": "8641",
               "doc_count": 107,
               "score": 13.884387742677438,
               "bg_count": 762
            },
            {
               "key": 58156,
               "key_as_string": "58156",
               "doc_count": 17,
               "score": 9.746428133759462,
               "bg_count": 28
            },
            {
               "key": 52973,
               "key_as_string": "52973",
               "doc_count": 95,
               "score": 9.65770100311672,
               "bg_count": 857
            },
            {
               "key": 35836,
               "key_as_string": "35836",
               "doc_count": 128,
               "score": 9.199001116457955,
               "bg_count": 1610
            }
         ]
 ...
```

① doc_count 展 了前景特征 里文 的数量。

② 个桶里面列出了聚合的 （例如，影片的ID）。

③ 桶内文 的数量 doc_count 。

④ 背景文 的数量，表示 在整个 背景里出 的 度。

可以看到， 得的第一个桶是 *Talladega Nights* 。它可以在所有 271 个文 中 到， 并不意外。 我 看下一个桶： 52245 。

个 ID 影片 *Blades of Glory*（ 誉之刃），它是一部 于男子学 滑 的喜 ，也是由 Will Ferrell 主演。可以看到喜 *Talladega Nights* 的用 它的推 是 59 次。 也意味着 21% 的前景特征用 推 了影片 *Blades of Glory* （ `59 / 271 = 0.2177` ）。

形成 比的是， *Blades of Glory* 在整个数据集合中 被推 了 185 次， 只占 0.26% （ `185 / 69796 = 0.00265` ）。因此 *Blades of Glory* 是一个 常：它在喜 *Talladega Nights* 的用 中是 著的共性（注：uncommonly common）。 就 到了一个好的推 ！

如果看完整的列表，它 都是好的喜 推 （其中很多也是由 Will Ferrell 主演）：

1. Blades of Glory（ 誉之刃）
2. Anchorman: The Legend of Ron Burgundy（王牌播音 ）
3. Semi-Pro（半 手）
4. Knocked Up（一夜大肚）
5. 40-Year-Old Virgin, The（四十 的老 男）

只是 significant_terms 它 大的一个示例，一旦 始使用 significant_terms ，可能 到 的情况，我 不想要最流行的，而想要 著的共性（注：uncommonly common）。 个 的聚合可以 示出一些数据里出人意料的 。

Doc Values and Fielddata

Doc Values

聚合使用一个叫 *doc values* 的数据 （在 [docvalues-intro] 里 介 ）。 Doc values 可以使聚合更快、更高效并且内存友好，所以理解它的工作方式十分有益。

Doc values 的存在是因 倒排索引只 某些操作是高效的。 倒排索引的 在于 包含某个 的文 ，而 于从 外一个方向的相反操作并不高效，即： 定 些 是否存在 个文 里，聚合需要 次 的 模式。

于以下倒排索引：

```
Term      Doc_1   Doc_2   Doc_3
-----------------------------------
brown   |   X   |   X   |
dog     |   X   |       |   X
dogs    |       |   X   |   X
fox     |   X   |       |   X
foxes   |       |   X   |
in      |       |   X   |
jumped  |   X   |       |   X
lazy    |   X   |   X   |
leap    |       |   X   |
over    |   X   |   X   |   X
quick   |   X   |   X   |   X
summer  |       |   X   |
the     |   X   |       |   X
-----------------------------------
```

如果我 想要 得所有包含 brown 的文 的 的完整列表，我 会 建如下 ：

```
GET /my_index/_search
{
  "query" : {
    "match" : {
      "body" : "brown"
    }
  },
  "aggs" : {
    "popular_terms": {
      "terms" : {
        "field" : "body"
      }
    }
  }
}
```

部分 又高效。倒排索引是根据 来排序的，所以我 首先在 列表中 到 brown ，然后描所有列， 到包含 brown 的文 。我 可以快速看到 Doc_1 和 Doc_2 包含 brown 个 token。

然后， 于聚合部分，我 需要 到 Doc_1 和 Doc_2 里所有唯一的 。用倒排索引做 件事情代 很高：我 会迭代索引里的 个 并收集 Doc_1 和 Doc_2 列里面 token。 很慢而且 以 展：随着 和文的数量 加， 行 也会 加。

Doc values 通 置 者 的 系来解决 个 。倒排索引将 映射到包含它 的文 ，doc values 将文 映射到它 包含的 ：

```
Doc      Terms
-----------------------------------------------------------------
Doc_1 | brown, dog, fox, jumped, lazy, over, quick, the
Doc_2 | brown, dogs, foxes, in, lazy, leap, over, quick, summer
Doc_3 | dog, dogs, fox, jumped, over, quick, the
-----------------------------------------------------------------
```

当数据被 置之后，想要收集到 Doc_1 和 Doc_2 的唯一 token 会非常容易。 得 个文 行，取所有的 ，然后求 个集合的并集。

因此，搜索和聚合是相互 密 的。搜索使用倒排索引 文 ，聚合操作收集和聚合 doc values 里的数据。

NOTE

Doc values 不 可以用于聚合。 任何需要 某个文 包含的 的操作都必 使用它。除了聚合， 包括排序， 字段 的脚本，父子 系 理（参 [parent-child] ）。

深入文

在上一 一 我 就 文 （doc values） 是 "更快、更高效并且内存友好" 。 听起来好像是不的 ，不 回来文 到底是如何工作的 ？

文 是在索引 与倒排索引同 生的。也就是 文 是按段来 生的并且是不可 的，正如用于搜索的倒排索引一 。同 ，和倒排索引一 ，文 也序列化到磁 。 些 于性能和伸 性很重要。

通 序列化一个持久化的数据 到磁 ，我 可以依 于操作系 的 存来管理内存，而不是在 JVM 堆 里 留数据。 当 "工作集（working set）" 数据要小于系 可用内存的情况下，操作系会自然的将文 留在内存， 将会 来和直接使用 JVM 堆 数据 相同的性能。

不 ，如果 的工作集 大于可用内存，操作系 会 始根据需要 文 行分 / 。 会 著慢于内存 留的数据 ，当然，它也 有使用 大于服 器内存容量的伸 性的好 。 如果 些数据 是粹的存 于 JVM 堆内存，那 唯一的 只能是随着内存溢出（OutOfMemory）而崩 （或是一个分 模式，正如操作系 的那 ）。

NOTE

因 文 不是由 JVM 来管理，所以 Elasticsearch 服 器可以配置一个很小的 JVM 堆 。 会 操作系 来更多的内存来做 存。同 也 来一个好 就是 JVM 的回收器工作在一个很小的堆 ， 果就是更快更高效的回收周期。

上，我 会建 分配机器内存的 50% 来 JVM 堆 。随着文 的引入， 个建始不再 用。 在 64gb 内存的机器上，也 可以考 堆 分配 4-16gb 的内存，而不是之前建 的 32gb。

有 更 的 ， 看 [heap-sizing].

列式存 的

从广 来 ，文 本 上是一个序列化的 列式存 。 正如我 上一 所 的，列式存 擅 某些操作，因 些数据的存 天然 合 些 。

而且，他 也同 擅 数据 ，特 是数字。 于 省磁 空 和快速 很重要。 代 CPU 的 理速度要比磁 快几个数量 （尽管即将到来的 NVMe 器正在迅速 小差距）。 意味着 少必 从磁 取的数据量 是有益的，尽管需要 外的 CPU 算来 行解 。

要了解它如何 助 数据，来看一 数字 型的文 ：

```
Doc      Terms
-----------------------------------------------------------------
Doc_1 | 100
Doc_2 | 1000
Doc_3 | 1500
Doc_4 | 1200
Doc_5 | 300
Doc_6 | 1900
Doc_7 | 4200
-----------------------------------------------------------------
```

按列布局意味着我 有一个 的数据 ： `[100,1000,1500,1200,300,1900,4200]` 。因 我 已 知道他 都是数字（而不是像文 或行中看到的 集合），所以我 可以使用 一的偏移来将他 排列。

而且， 的数字有很多 技巧。 会注意到 里 个数字都是 100 的倍数，文 会 一个段里面的所有数 ，并使用一个 最大公 数，方便做 一 的数据 。

如果我 保存 `100` 作 此段的除数，我 可以 个数字都除以 100，然后得到： `[1,10,15,12,3,19,42]` 。 在 些数字 小了，只需要很少的位就可以存 下，也 少了磁 存放的大小。

文 正是使用了像 的一些技巧。它会按依次 以下 模式:

1. 如果所有的数 各不相同（或 失）， 置一个 并 些
2. 如果 些 小于 256，将使用一个 的 表
3. 如果 些 大于 256， 是否存在一个最大公 数
4. 如果没有存在最大公 数，从最小的数 始， 一 算偏移量 行

会 些 模式不是 的通用的 方式，比如 DEFLATE 或是 LZ4。 因 列式存 的 是 格且良好定 的，我 可以通 使用 的模式来 到比通用 算法（如 LZ4 ）更高的 效果。

NOTE

也 会想 "好 ，貌似 数字很好，不知道字符串 ？" 通 借助 序表（ordinal table），字符 型也是 似 行 的。字符 型是去重之后存放到 序表的，通 分配 一个 ID，然后 些 ID 和数 型的文 一 使用。 也就是 ，字符 型和数型一 有相同的 特性。

序表本身也有很多 技巧，比如固定 度、 或是前 字符 等等。

禁用文

文 所有字段 用，除了分析字符 型字段。也就是 所有的数字、地理坐 、日期、IP 和不分析（ `not_analyzed` ）字符 型。

分析字符 型 不使用文 。分析流程会 生很多新的 token， 会 文 不能高效的工作。我 将在 聚合与分析 如何使用分析字符 型来做聚合。

因 文 用， 可以 数据集里面的大多数字段 行聚合和排序操作。但是如果 知道 永 也不会 某些字段 行聚合、排序或是使用脚本操作？

尽管 ，但当 些情况出 ， 是希望有 法来 特定的字段禁用文 。 回 省磁 空 （因 文 再也没有序列化到磁 ），也 能提升些 索引速度（因 不需要生成文 ）。

要禁用文 ，在字段的映射（mapping） 置 `doc_values: false` 即可。例如， 里我 建了一个新的索引，字段 `"session_id"` 禁用了文 ：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "session_id": {
          "type":       "string",
          "index":      "not_analyzed",
          "doc_values": false ①
        }
      }
    }
  }
}
```

① 通 置 `doc_values: false`， 个字段将不能被用于聚合、排序以及脚本操作

反 来也是可以 行配置的： 一个字段可以被聚合，通 禁用倒排索引，使它不能被正常搜索，例如：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "customer_token": {
          "type":       "string",
          "index":      "not_analyzed",
          "doc_values": true, ①
          "index": "no" ②
        }
      }
    }
  }
}
```

① 文 被 用来允 聚合

② 索引被禁用了， 字段不能被 /搜索

通　置 doc_values: true 和 index: no ，我　得到一个只能被用于聚合/排序/脚本的字段。无可否　，　是一个非常　的需求，但有　很有用。

聚合与分析

有些聚合，比如 terms 桶，　操作字符串字段。字符串字段可能是 analyzed 或者 not_analyzed ，　那　来了，分析是　影　聚合的　？

答案是影　“很多”，有　个原因：分析影　聚合中使用的　tokens　，并且　doc　values　不能使用于分析字符串。

　我　解决第一个　：分析 tokens 的　生如何影　聚合。首先索引一些代表美国各个州的文　：

```
POST /agg_analysis/data/_bulk
{ "index": {}}
{ "state" : "New York" }
{ "index": {}}
{ "state" : "New Jersey" }
{ "index": {}}
{ "state" : "New Mexico" }
{ "index": {}}
{ "state" : "New York" }
{ "index": {}}
{ "state" : "New York" }
```

我　希望　建一个数据集里各个州的唯一列表，并且　数。　，　我　使用 terms 桶：

```
GET /agg_analysis/data/_search
{
    "size" : 0,
    "aggs" : {
        "states" : {
            "terms" : {
                "field" : "state"
            }
        }
    }
}
```

得到　果：

```
{
...
   "aggregations": {
      "states": {
         "buckets": [
            {
               "key": "new",
               "doc_count": 5
            },
            {
               "key": "york",
               "doc_count": 3
            },
            {
               "key": "jersey",
               "doc_count": 1
            },
            {
               "key": "mexico",
               "doc_count": 1
            }
         ]
      }
   }
}
```

宝 儿， 完全不是我 想要的！没有 州名 数，聚合 算了 个 的数目。背后的原因很 ：聚合是基于倒排索引 建的，倒排索引是 后置分析（*post-analysis*）的。

当我 把 些文 加入到 Elasticsearch 中 ，字符串 `"New York"` 被分析/分析成 `["new", "york"]` 。些 独的 tokens，都被用来填充聚合 数，所以我 最 看到 `new` 的数量而不是 `New York` 。

然不是我 想要的行 ，但幸 的是很容易修正它。

我 需要 state 定 multifield 并且 置成 `not_analyzed` 。 可以防止 `New York` 被分析，也意味着在聚合 程中它会以 个 token 的形式存在。 我 完整的 程，但 次指定一个 *raw* multifield：

```
DELETE /agg_analysis/
PUT /agg_analysis
{
  "mappings": {
    "data": {
      "properties": {
        "state" : {
          "type": "string",
          "fields": {
            "raw" : {
              "type": "string",
              "index": "not_analyzed"①
            }
          }
        }
      }
    }
  }
}

POST /agg_analysis/data/_bulk
{ "index": {}}
{ "state" : "New York" }
{ "index": {}}
{ "state" : "New Jersey" }
{ "index": {}}
{ "state" : "New Mexico" }
{ "index": {}}
{ "state" : "New York" }
{ "index": {}}
{ "state" : "New York" }

GET /agg_analysis/data/_search
{
  "size" : 0,
  "aggs" : {
    "states" : {
        "terms" : {
            "field" : "state.raw" ②
        }
    }
  }
}
```

① 次我 式映射 state 字段并包括一个 not_analyzed 字段。

② 聚合 state.raw 字段而不是 state 。

在 行聚合，我 得到了合理的 果：

```
{
...
   "aggregations": {
      "states": {
         "buckets": [
            {
               "key": "New York",
               "doc_count": 3
            },
            {
               "key": "New Jersey",
               "doc_count": 1
            },
            {
               "key": "New Mexico",
               "doc_count": 1
            }
         ]
      }
   }
}
```

在 中， 的 很容易被察 ，我 的聚合会返回一些奇怪的桶，我 会 住分析的 。 之，很少有在聚合中使用分析字段的 例。当我 疑惑 ，只要 加一个 multifield 就能有 。

分析字符串和 Fielddata（Analyzed strings and Fielddata）

当第一个 及如何聚合数据并 示 用 ，第二个 主要是技 和幕后。

Doc values 不支持 `analyzed` 字符串字段，因 它 不能很有效的表示多 字符串。 Doc values 最有效的是，当 个文 都有一个或几个 tokens ， 但不是无数的，分析字符串（想象一个 PDF ，可能有几兆字 并有数以千 的独特 tokens）。

出于 个原因，doc values 不生成分析的字符串，然而， 些字段 然可以使用聚合，那 可能 ？

答案是一 被称 *fielddata* 的数据 。与 doc values 不同，fielddata 建和管理 100% 在内存中，常 于 JVM 内存堆。 意味着它本 上是不可 展的，有很多 情况下要提防。本章的其余部分是解决在分析字符串上下文中 fielddata 的挑 。

NOTE	从 史上看，fielddata 是 所有 字段的 置。但是 Elasticsearch 已 移到 doc values 以 少 OOM 的几率。分析的字符串是 然使用 fielddata 的最后一 地。 最 目 是建立一个序列化的数据 似于 doc values ，可以 理高 度的分析字符串，逐 淘汰 fielddata。

高基数内存的影 （High-Cardinality Memory Implications）

避免分析字段的 外一个原因就是：高基数字段在加 到 fielddata 会消耗大量内存。 分析的 程会 常（尽管不 是 ）生成大量的 token， 些 token 大多都是唯一的。 会 加字段的整体基数并且 来更大的内存 力。

有些 型的分析 于内存来 度 不友好，想想 n-gram 的分析 程， New York 会被 n-gram 分析成以下 token：

- ne
- ew
- w{nbsp}
- {nbsp}y
- yo
- or
- rk

可以想象 n-gram 的 程是如何生成大量唯一 token 的，特 是在分析成段文本的 候。当 些数据加 到内存中，会 而易 的将我 堆空 消耗殆尽。

因此，在聚合字符串字段之前， 估情况：

- 是一个 not_analyzed 字段 ？如果是，可以通 doc values 省内存 。
- 否 ， 是一个 analyzed 字段，它将使用 fielddata 并加 到内存中。 个字段因 ngrams 有一个非常大的基数？如果是， 于内存来 度不友好。

限制内存使用

一旦分析字符串被加 到 fielddata ，他 会一直在那里，直到被 逐（或者 点崩 ）。由于 个原因，留意内存的使用情况，了解它是如何以及何 加 的， 限制 集群的影 是很重要的。

Fielddata 是 延 加 。如果 从来没有聚合一个分析字符串，就不会加 fielddata 到内存中。此外，fielddata 是基于字段加 的， 意味着只有很活 地使用字段才会 加 fielddata 的 担。

然而， 里有一个令人 的地方。假 的 是高度 性和只返回命中的 100 个 果。大多数人 fielddata 只加 100 个文 。

情况是，fielddata 会加 索引中（ 特定字段的） 所有的 文 ，而不管 的特 性。是 ：如果 会 文 X、Y 和 Z，那很有可能会在下一个 中 其他文 。

与 doc values 不同，fielddata 不会在索引 建。相反，它是在 行 ， 填充。 可能是一个比 的操作，可能需要一些 。 将所有的信息一次加 ，再将其 持在内存中的方式要比反 只加 一个 fielddata 的部分代 要低。

JVM 堆 是有限 源的， 被合理利用。 限制 fielddata 堆使用的影 有多套机制， 些限制方式非常重要，因 堆 的乱用会 致 点不 定（感 慢的 回收机制），甚至 致 点宕机（通常伴随 OutOfMemory 常）。

堆大小（Choosing a Heap Size）

在 置 Elasticsearch 堆大小 需要通 `$ES_HEAP_SIZE` 境 量 用 个 ：

不要超 可用 *RAM* 的 *50%*

Lucene 能很好利用文件系 的 存，它是通 系 内核管理的。如果没有足 的文件系存空 ，性能会受到影 。 此外， 用于堆的内存越多意味着其他所有使用 doc values 的字段内存越少。

不要超 *32 GB*

如果堆大小小于 32 GB，JVM 可以利用指 ， 可以大大降低内存的使用： 个指 4 字 而不是 8 字 。

更 和更完整的堆大小 ， 参 [heap-sizing]

Fielddata的大小

`indices.fielddata.cache.size` 控制 fielddata 分配的堆空 大小。 当 起一个 ，分析字符串的聚合将会被加 到 fielddata，如果 些字符串之前没有被加 。如果 果中 fielddata 大小超 了指定 大小，其他的 将会被回收从而 得空 。

情况下， 置都是 *unbounded*，Elasticsearch 永 都不会从 fielddata 中回收数据。

个 置是刻意 的：fielddata 不是 存。它是 留内存里的数据 ，必 可以快速 行 ，而且 建它的代 十分高昂。如果 个 求都重 数据，性能会十分糟 。

一个有界的大小会 制数据 回收数据。我 会看何 置 个 ，但 首先 以下警告：

WARNING

个 置是一个安全 士，而非内存不足的解决方案。

如果没有足 空 可以将 fielddata 保留在内存中，Elasticsearch 就会 刻从磁重 数据，并回收其他数据以 得更多空 。内存的回收机制会 致重度磁 I/O，并且在内存中生成很多 ， 些 必 在 些 候被回收掉。

想我 正在 日志 行索引， 天使用一个新的索引。通常我 只 去一 天的数据感 趣，尽管我会保留老的索引，但我 很少需要 它 。不 如果采用 置，旧索引的 fielddata 永 不会从 存中回收！fieldata 会保持 直到 fielddata 生断熔（ 参 断路器）， 我 就无法 入更多的 fielddata。

个 候，我 被困在了死胡同。但我 然可以 旧索引中的 fielddata，也无法加 任何新的 。相反，我 回收旧的数据，并 新 得更多空 。

了防止 生 的事情，可以通 在 `config/elasticsearch.yml` 文件中 加配置 fielddata 置一个上限：

```
indices.fielddata.cache.size:  20% ①
```

① 可以 置堆大小的百分比，也可以是某个 ，例如： `5gb`。

有了 个 置，最久未使用（LRU）的 fielddata 会被回收 新数据 出空 。

WARNING

可能 在 文 有 外一个 置： `indices.fielddata.cache.expire` 。

个 置 永 都不会 被使用！它很有可能在不久的将来被 用。

个 置要求 Elasticsearch 回收那些 期 的 fielddata，不管 些 有没有被用到。

性能是件 很糟 的事情。回收会有消耗性能，它刻意的安排回收方式，而没能得任何回 。

没有理由使用 个 置：我 不能从理 上假 一个有用的情形。目前，它的存在只是 了向前兼容。我 只在很有以前提到 个 置，但不幸的是 上各 文章都将其作 一 性能 的小 来推 。

它不是。永 不要使用！

控 fielddata（Monitoring fielddata）

无 是仔 控 fielddata 的内存使用情况， 是看有无数据被回收都十分重要。高的回收数可以 示 重的 源 以及性能不佳的原因。

Fielddata 的使用可以被 控：

- 按索引使用 `indices-stats` API ：

```
GET /_stats/fielddata?fields=*
```

- 按 点使用 {ref}/cluster-nodes-stats.html[`nodes-stats` API] ：

```
GET /_nodes/stats/indices/fielddata?fields=*
```

- 按索引 点：

```
GET /_nodes/stats/indices/fielddata?level=indices&fields=*
```

使用 置 `?fields=*` ，可以将内存使用分配到 个字段。

断路器

机敏的 者可能已 fielddata 大小 置的一个 。fielddata 大小是在数据加 之后 的。如果一个 加 比可用内存更多的信息到 fielddata 中会 生什 ？答案很丑 ：我 会 到 OutOfMemoryException 。

Elasticsearch 包括一个 *fielddata* 断熔器 ， 个 就是 了 理上述情况。 断熔器通 内部 （字段的 型、基数、大小等等）来估算一个 需要的内存。它然后 要求加 的 fielddata 是否会 致 fielddata 的 量超 堆的配置比例。

如果估算　的大小超出限制，就会　触　断路器，　会被中止并返回　常。　都　生在数据加　之前，也就意味着不会引起 OutOfMemoryException 。

可用的断路器（Available Circuit Breakers）

Elasticsearch 有一系列的断路器，它　都能保　内存不会超出限制：

`indices.breaker.fielddata.limit`
　`fielddata` 断路器　置堆的 60% 作　fielddata 大小的上限。

`indices.breaker.request.limit`
　`request`　断路器估算需要完成其他　求部分的　大小，例如　建一个聚合桶，限制是堆内存的 40%。

`indices.breaker.total.limit`
　`total` 揉合 `request` 和 `fielddata` 断路器保　者　合起来不会使用超　堆内存的 70%。

断路器的限制可以在文件 `config/elasticsearch.yml` 中指定，可以　更新一个正在　行的集群：

```
PUT /_cluster/settings
{
  "persistent" : {
    "indices.breaker.fielddata.limit" : "40%" ①
  }
}
```

① 个限制是按　内存的百分比　置的。

最好　断路器　置一个相　保守点的　。　住　fielddata　需要与　`request`　断路器共享堆内存、索引　冲内存和　器　存。Lucene　的数据被用来　造索引，以及各　其他　的数据　。正因如此，它　非常保守，只有　60%　。　于　的　置可能会引起潜在的堆　溢出（OOM）　常，会使整个　点宕掉。

一方面，　度保守的　只会返回　常，　用程序可以　常做相　理。　常比服　器崩　要好。些　常　也能促　我　行重新　估：　什　个　需要超　堆内存的 60% 之多？

TIP

在 Fielddata的大小 中，我　提　于　fielddata 的大小加一个限制，从而　保旧的无用 fielddata　被回收的方法。　`indices.fielddata.cache.size`　和 `indices.breaker.fielddata.limit`　之　的　系非常重要。　如果断路器的限制低于　存大小，没有数据会被回收。　了能正常工作，断路器的限制 必　要比　存大小要高。

得注意的是：断路器是根据　堆内存大小估算　大小的，而　非　根据　堆内存的使用情况。　是由于各　技　原因造成的（例如，堆可能看上去是　的但　上可能只是在等待　回收，　使我以　行合理的估算）。但作　端用　，　意味着　置需要保守，因　它是根据　堆内存必要的，而　不是可用堆内存。

Fielddata 的

想我 正在 行一个 站允 用 收听他 喜 的歌曲。 了 他 可以更容易的管理自己的音，用 可以 歌曲 置任何他 喜 的 ， 我 就会有很多歌曲被附上 rock（ ） 、 hiphop（哈） 和 electronica（ 音） ，但也会有些歌曲被附上 my_16th_birthday_favorite_anthem 的 。

在 想我 想要 用 展示 首歌曲最受 迎的三个 ，很有可能 rock 的 会排在三个中的最前面，而 my_16th_birthday_favorite_anthem 不太可能得到 。 尽管如此，了 算最受 迎的 ，我 必 制将 些一次性使用的 加 到内存中。

感 fielddata ，我 可以控制 状况。我 知道 自己只 最流行的 感 趣，所以我 可以 地避免加 那些不太有意思的 尾 ：

```
PUT /music/_mapping/song
{
  "properties": {
    "tag": {
      "type": "string",
      "fielddata": { ①
        "filter": {
          "frequency": { ②
            "min":              0.01, ③
            "min_segment_size": 500  ④
          }
        }
      }
    }
  }
}
```

① fielddata 字允 我 配置 fielddata 理 字段的方式。

② frequency 器允 我 基于 率 加 fielddata。

③ 只加 那些至少在本段文 中出 1% 的 。

④ 忽略任何文 个数小于 500 的段。

有了 个映射，只有那些至少在 本段 文 中出 超 1% 的 才会被加 到内存中。我 也可以指定一个最大 ，它可以被用来排除 常用 ，比如 停用 。

情况下， 是按照段来 算的。 是 的一个限制：fielddata 是按段来加 的，所以可 的只是 段内的 率。但是， 个限制也有些有趣的特性：它可以 受 迎的新 迅速提升到 部。

比如一个新 格的歌曲在一夜之 受大 迎，我 可能想要将 新 格的歌曲 包括在最受 迎列表中，但如果我 倚 索引做完整的 算 取 ，我 就必 等到新 得像 rock 和 electronica）一 流行。 由于 度 的 方式，新加的 会很快作 高 出在新段内，也当然会迅速上升到 部。

min_segment_size 参数要求 Elasticsearch 忽略某个大小以下的段。 如果一个段内只有少量文 ，它的

会非常粗略没有任何意 。 小的分段会很快被合并到更大的分段中，某一刻超 个限制，将会被入 算。

TIP

通 次来 并不是唯一的 ，我 也可以使用正 式来决定只加 那些匹配的 。例如，我 可以用 `regex` 器 理 twitte 上的消息只将以 `#` 号 始的 加 到内存中。 假 我 使用的分析器会保留 点符号，像 `whitespace` 分析器。

Fielddata 内存使用有 巨大的 影 ， 衡也是 而易 的：我 上是在忽略数据。但 于很多用， 衡是合理的，因 些数据根本就没有被使用到。内存的 省通常要比包括一个大量而无用的尾 更 重要。

加 fielddata

Elasticsearch 加 内存 fielddata 的 行 是 延 加 。 当 Elasticsearch 第一次 某个字段 ，它将会完整加 个字段所有 Segment 中的倒排索引到内存中，以便于以后的 能 取更好的性能。

于小索引段来 ， 个 程的需要的 可以忽略。但如果我 有一些 5 GB 的索引段，并希望加 10 GB 的 fielddata 到内存中， 个 程可能会要数十秒。 已 秒 的用 很 会接受停 数秒着没反 的 站。

有三 方式可以解决 个延 高峰：

- 加 fielddata
- 加 全局序号
- 存

所有的 化都基于同一概念： 加 fielddata， 在用 行搜索 就不会 到延 高峰。

加 fielddata（Eagerly Loading Fielddata）

第一个工具称 加 （与 的 延 加 相 ）。随着新分段的 建（通刷新、写入或合并等方式）， 字段 加 可以使那些 搜索不可 的分段里的 fielddata 提前 加 。

就意味着首次命中分段的 不需要促 fielddata 的加 ，因 fielddata 已 被入到内存。避免了用 遇到搜索 的情形。

加 是按字段 用的，所以我 可以控制具体 个字段可以 先加 ：

```
PUT /music/_mapping/_song
{
  "tags": {
    "type": "string",
    "fielddata": {
      "loading" : "eager" ①
    }
  }
}
```

① 置 fielddata.loading: eager 可以告 Elasticsearch 先将此字段的内容 入内存中。

Fielddata 的 入可以使用 update-mapping API 已有字段 置 lazy 或 eager 模式。

WARNING

加 只是 的将 入 fielddata 的代 移到索引刷新的 候，而不是 ，从而大大提高了搜索体 。

体 大的索引段会比体 小的索引段需要更 的刷新 。通常，体 大的索引段是由那些已 可 的小分段合并而成的，所以 慢的刷新 也不是很重要。

全局序号（Global Ordinals）

有 可以用来降低字符串 fielddata 内存使用的技 叫做 序号 。

想我 有十 文 ， 个文 都有自己的 status 状 字段，状 共有三 ： status_pending 、 status_published 、 status_deleted 。如果我 个文 都保留其状 的完整字符串形式，那 个文 就需要使用 14 到 16 字 ，或 共 15 GB。

取而代之的是我 可以指定三个不同的字符串， 其排序、 号：0，1，2。

```
Ordinal | Term
------------------
0       | status_deleted
1       | status_pending
2       | status_published
```

序号字符串在序号列表中只存 一次， 个文 只要使用数 号的序号来替代它原始的 。

```
Doc     | Ordinal
------------------------
0       | 1  # pending
1       | 1  # pending
2       | 2  # published
3       | 0  # deleted
```

可以将内存使用从 15 GB 降到 1 GB 以下！

但 里有个 ， 得 fielddata 是按分 段 来 存的。如果一个分段只包含 个状 （ status_deleted 和 status_published ）。那 果中的序号（0 和 1）就会与包含所有三个状 的分段不一 。

如果我 status 字段 行 terms 聚合，我 需要 字符串的 行聚合，也就是 我需要 所有分段中相同的 。一个 粗暴的方式就是 个分段 行聚合操作，返回 个分段的字符串 ，再将它 得出完整的 果。 尽管 做可行，但会很慢而且大量消耗 CPU。

取而代之的是使用一个被称 全局序号 的 。 全局序号是一个 建在 fielddata 之上的数据 ，它只占用少量内存。唯一 是 跨所有分段 的，然后将它 存入一个序号列表中，正如我描述 的那 。

在， terms 聚合可以 全局序号 行聚合操作，将序号 成真 字符串 的 程只会在聚合 束 生一次。 会将聚合（和排序）的性能提高三到四倍。

建全局序号（**Building global ordinals**）

当然，天下没有免 的 餐。 全局序号分布在索引的所有段中，所以如果新 或 除一个分段 ，需要 全局序号 行重建。 重建需要 取 个分段的 个唯一 ，基数越高（即存在更多的唯一 ） 个 程会越 。

全局序号是 建在内存 fielddata 和 doc values 之上的。 上，它 正是 doc values 性能表 不 的一个主要原因。

和 fielddata 加 一 ，全局序号 也是延 建的。首个需要 索引内 fielddata 的 求会促 全局序号的 建。由于字段的基数不同， 会 致 用 来 著延 一糟 果。一旦全局序号 生重建， 会使用旧的全局序号，直到索引中的分段 生 化：在刷新、写入或合并之后。

建全局序号（**Eager global ordinals**）

个字符串字段 可以通 配置 先 建全局序号：

```
PUT /music/_mapping/_song
{
  "song_title": {
    "type": "string",
    "fielddata": {
      "loading" : "eager_global_ordinals" ①
    }
  }
}
```

① 置 eager_global_ordinals 也暗示着 fielddata 是 加 的。

正如 fielddata 的 加 一 ， 建全局序号 生在新分段 于搜索可 之前。

NOTE

序号的 建只被 用于字符串。数 信息（integers（整数）、geopoints（地理 度）、dates（日期）等等）不需要使用序号映射，因 些 自己本 上就是序号映射。

因此，我 只能 字符串字段 建其全局序号。

也可以 Doc values 行全局序号 建：

```
PUT /music/_mapping/_song
{
  "song_title": {
    "type":       "string",
    "doc_values": true,
    "fielddata": {
      "loading" : "eager_global_ordinals" ①
    }
  }
}
```

① 情况下，fielddata 没有 入到内存中，而是 doc values 被 入到文件系 存中。

与 fielddata 加 不一 ， 建全局序号会 数据的 性 生影 ， 建一个高基数的全局序号会使一个刷新延 数秒。 在于是 次刷新 付出代 ， 是在刷新后的第一次 。如果 常索引而 少，那 在 付出代 要比 次刷新 要好。如果写大于 ，那 在 在 重建全局序号将会是一个更好的 。

TIP 景 化全局序号的重建 次。如果我 有高基数字段需要花数秒 重建， 加 `refresh_interval` 的刷新的 从而可以使我 的全局序号保留更 的有效期， 也会 省 CPU 源，因 我 重建的 次下降了。

索引 器（Index Warmers）

最后我 索引 器 。 器早于 fielddata 加 和全局序号 加 之前出 ，它 然尤其存在的理由。一个索引 器允 我 指定一个 和聚合 要在新分片 于搜索可 之前 行 。 个想法是通 先填充或 存 用 永 无法遇到延 的波峰。

原来， 器最重要的用法是 保 fielddata 被 先加 ，因 通常是最耗 的一 。 在可以通前面 的那些技 来更好的控制它，但是 器 是可以用来 建 器 存，当然我 也 是能 用它来 加 fielddata。

我 注 一个 器然后解 生了什 ：

```
PUT /music/_warmer/warmer_1 ①
{
  "query" : {
    "bool" : {
      "filter" : {
        "bool": {
          "should": [ ②
            { "term": { "tag": "rock"        }},
            { "term": { "tag": "hiphop"      }},
            { "term": { "tag": "electronics" }}
          ]
        }
      }
    }
  },
  "aggs" : {
    "price" : {
      "histogram" : {
        "field" : "price", ③
        "interval" : 10
      }
    }
  }
}
```

① 器被 到索引（music）上，使用接入口 _warmer 以及 ID （warmer_1）。

② 三 最受 迎的曲 建 器 存。

③ 字段 price 的 fielddata 和全局序号会被 加 。

器是根据具体索引注 的， 个 器都有唯一的 ID，因 个索引可能有多个 器。

然后我 可以指定 ，任何 。它可以包括 、 器、聚合、排序 、脚本，任何有效的 表式都 不夸 。 里的目的是想注 那些可以代表用 生流量 力的 ，从而将合 的内容 入 存。

当新建一个分段 ，Elasticsearch 将会 行注 在 器中的 。 行 些 会 制加 存，只有在所有 器 行完， 个分段才会 搜索可 。

WARNING

与 加 似， 器只是将冷 存的代 移到刷新的 候。当注 器 ，做出明智的决定十分重要。 了 保 个 存都被 入，我 可以 加入上千的 器，但 也会使新分段 于搜索可 的 急 上升。

中，我 会 少量代表大多数用 的 ，然后注 它 。

有些管理的 （比如 得已有 器和 除 器）没有在本小 提到，剩下的 内容可以参考 {ref}/indices-warmers.html[器文 （warmers documentation）]。

化聚合

“elasticsearch 里面桶的叫法和 SQL 里面分 的概念是 似的，一个桶就 似 SQL 里面的一个 group，多 嵌套的 aggregation， 似 SQL 里面的多字段分 （group by field1,field2, ...），注意 里 是概念 似，底 的 原理是不一 的。 — 者注”

terms 桶基于我 的数据 建桶；它并不知道到底生成了多少桶。 大多数 候 个字段的聚合 是非常快的， 但是当需要同 聚合多个字段 ，就可能会 生大量的分 ，最 果就是占用 es 大量内存，从而 致 OOM 的情况 生。

假 我 在有一些 于 影的数据集， 条数据里面会有一个数 型的字段存 表演 影的所有演 的名字。

```
{
  "actors" : [
    "Fred Jones",
    "Mary Jane",
    "Elizabeth Worthing"
  ]
}
```

如果我 想要 出演影片最多的十个演 以及与他 合作最多的演 ，使用聚合是非常 的：

```
{
  "aggs" : {
    "actors" : {
      "terms" : {
         "field" : "actors",
         "size" :  10
      },
      "aggs" : {
        "costars" : {
          "terms" : {
            "field" : "actors",
            "size" :  5
          }
        }
      }
    }
  }
}
```

会返回前十位出演最多的演 ，以及与他 合作最多的五位演 。 看起来是一个 的聚合 ，最 只返回 50 条数据！

但是， 个看上去 的 可以 而易 地消耗大量内存，我 可以通 在内存中 建一个 来 看 个 terms 聚合。 actors 聚合会 建 的第一 ， 个演 都有一个桶。然后，内套在第一 的 个 点之下， costar 聚合会 建第二 ， 个 合出演一个桶， 参 Build full depth tree 所示。 意味着

部影片会生成 n^2 个桶！

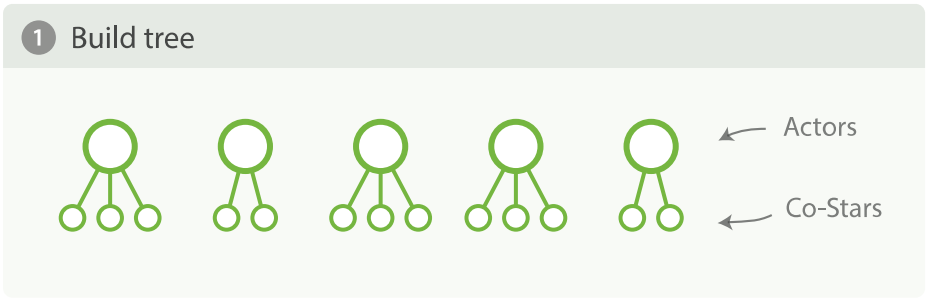

Figure 8. Build full depth tree

用真 点的数据， 想平均 部影片有 10 名演 ， 部影片就会生成 10^2 == 100 个桶。如果 共有 20 ，000 部影片，粗率 算就会生成 2，000，000 个桶。

在， 住，聚合只是 的希望得到前十位演 和与他 合出演者， 共 50 条数据。 了得到最的 果，我 建了一个有 2，000，000 桶的 ，然后 其排序，取 top10。 Sort tree 和 Prune tree 个 程 行了 述。

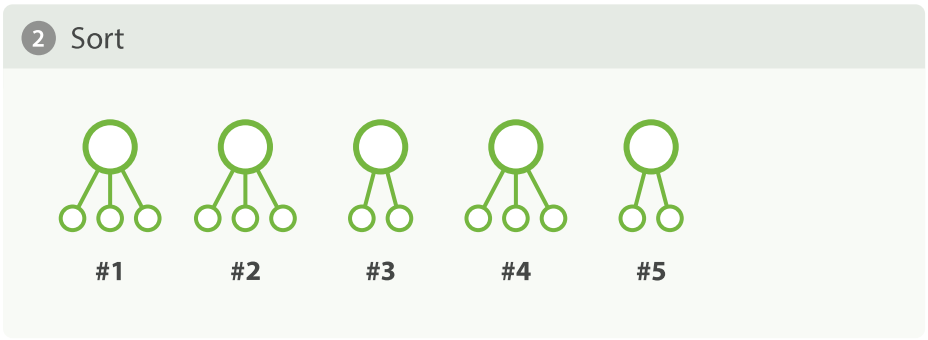

Figure 9. Sort tree

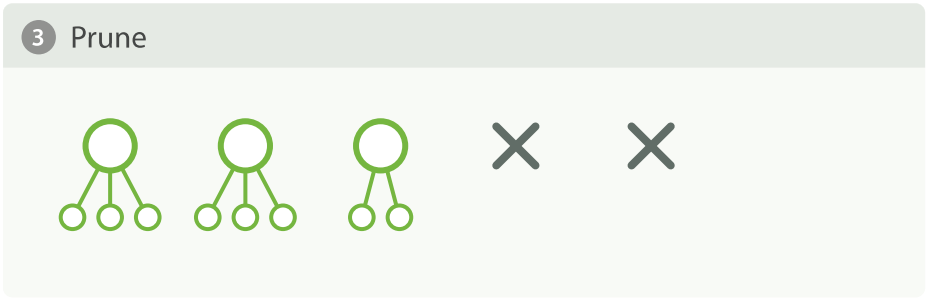

Figure 10. Prune tree

我 一定非常 狂，在 2 万条数据下 行任何聚合 都是 无 力的。如果我 有 2 文 ，想要得到前 100 位演 以及与他 合作最多的 20 位演 ，作 的最 果会出 什 情况 ？

可以推 聚合出来的分 数非常大，会使 策略 以 持。世界上并不存在足 的内存来支持 不受控制的聚合 。

深度　先与广度　先（Depth-First Versus Breadth-First）

Elasticsearch 允　我　改　聚合的 集合模式 ，就是　了　　状况。 我　之前展示的策略叫做 深度　先　　，它是　　置，　　先　建完整的　，然后修剪无用　点。　　深度　先　　的方式　于大多数聚合都能正常工作，但　于如我　演　和　合演　　例子的情形就不太　用。

　了　　些特殊的　用　景，我　　使用　一　集合策略叫做　　广度　先　　。　策略的工作方式有些不同，它先　行第一　聚合，　再　　下一　聚合之前会先做修剪。 Build first level 和　Prune first level　　个　程　行了　述。

在我　的示例中，　`actors`　聚合会首先　行，在　个　候，我　的　只有一　，但我　已　知道了前　10 位的演　！　就没有必要保留其他的演　信息，因　它　无　如何都不会出　在前十位中。

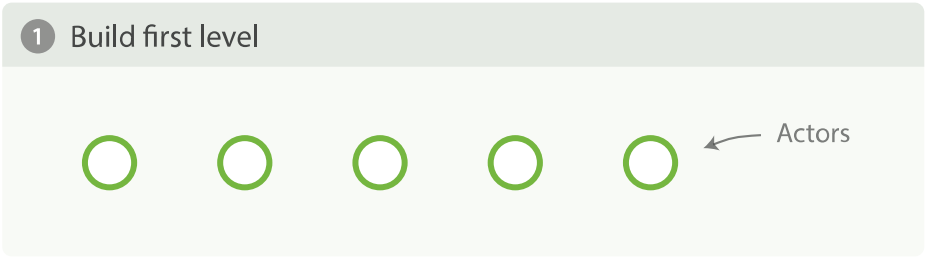

Figure 11. Build first level

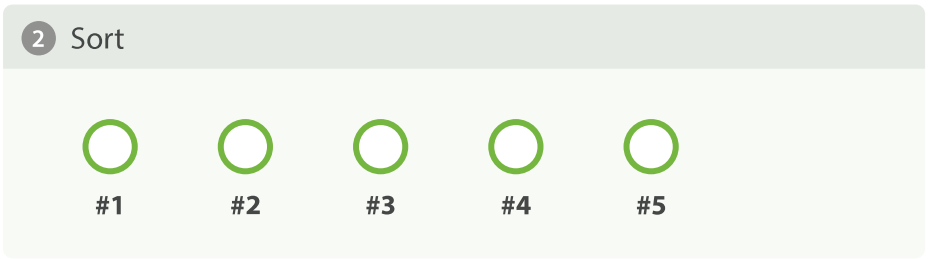

Figure 12. Sort first level

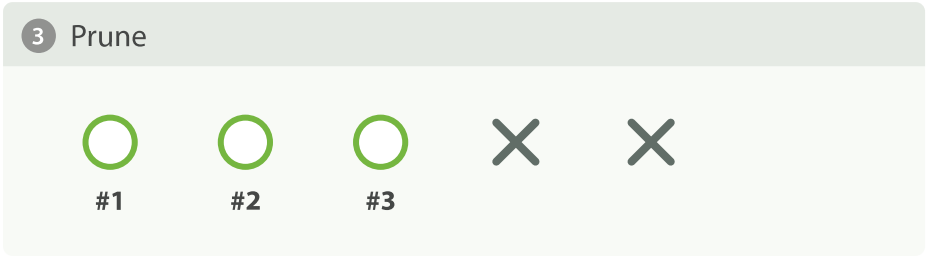

Figure 13. Prune first level

因　我　已　知道了前十名演　，我　可以安全的修剪其他　点。修剪后，下一　是基于　它的　　行模式

入的，重　　行　个　程直到聚合完成，如　　Populate full depth for remaining nodes 所示。
　　景下，广度　先可以大幅度　省内存。

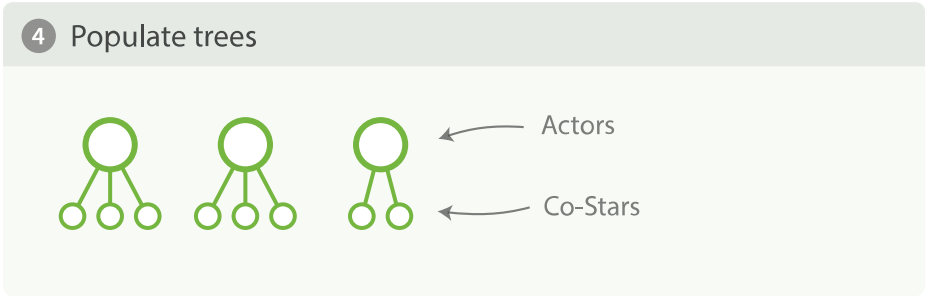

Figure 14. Populate full depth for remaining nodes

要使用广度　先，只需　　的通　参数 `collect`　　：

```
{
  "aggs" : {
    "actors" : {
      "terms" : {
         "field" :        "actors",
         "size" :         10,
         "collect_mode" : "breadth_first" ①
      },
      "aggs" : {
        "costars" : {
          "terms" : {
            "field" : "actors",
            "size" :  5
          }
        }
      }
    }
  }
}
```

① 按聚合来　　`breadth_first`。

广度　先　　用于　个　的聚合数量　　小于当前　　数的情况下，因　广度　先会在内存中　存裁剪后的　　需要　存的　个　的所有数据，以便于它的子聚合分　　可以　用上　聚合的数据。

广度　先的内存使用情况与裁剪后的　存分　数据量是成　性的。　于很多聚合来　，　个桶内的文　数量是相当大的。　　想象一　按月分　的直方　，　　数肯定是固定的，因　　年只有12个月，　个　候　个月下的数据量可能非常大。　使广度　先不是一个好的　　，　也是　什　深度　先作　　策略的原因。

　　上面演　的例子，如果数据量越大，那　　的使用深度　先的聚合模式生成的　分　数就会非常多，但是　估二　的聚合字段分　后的数据量相比　的分　数会小很多所以　　情况下使用广度　先的模式能大大　省内存，从而通　　化聚合模式来大大提高了在某些特定　景下聚合　　的成功率。

本小 涵 了 多基本理 以及很多深入的技 。聚合 Elasticsearch 来了 以言 的大能力和 活性。桶与度量的嵌套能力，基数与百分位数的快速估算能力，定位信息中 常的能力，所有的 些都在近乎 的情况下操作的，而且全文搜索是并行的，它 改 了很多企 的游 。

聚合是 一 功能特性：一旦我 始使用它，我 就能 到很多其他的可用 景。 表与分析 于很多 来 都是核心功能（无 是 用于商 智能 是服 器日志）。

Elasticsearch 大多数 字段 用 doc values，所以在一些搜索 景大大的 省了内存使用量，但是需要注意的是只有不分 的 string 型的字段才能使用 特性。

内存的管理形式可以有多 形式， 取决于我 特定的 用 景：

- 在 ， 好数据，使聚合 行在 `not_analyzed` 字符串而不是 `analyzed` 字符串，可以有效的利用 doc values 。
- 在 ， 分析 不会在之后的聚合 算中 建高基数字段。
- 在搜索 ，合理利用近似聚合和数据 。
- 在 点 ， 置硬内存大小以及 的断熔限制。
- 在 用 ，通 控集群内存的使用情况和 Full GC 的 生 率，来 整是否需要 集群源添加更多的机器 点

大多数 施会 用到以上一 或几 方法。 切的 合方式与我 特定的系 境高度相 。

无 采取何 方式， 于 有的 行 估，并同 建短期和 期 ，都十分重要。先决定当前内存的使用情况和需要做的事情（如果有），通 估数据 速度，来决定未来半年或者一年的集群的 ，使用何 方式来 展。

最好在建立集群之前就 好 些内容，而不是在我 集群堆内存使用 90% 的 候再 抱佛脚。