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。

NOTE

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。

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 。

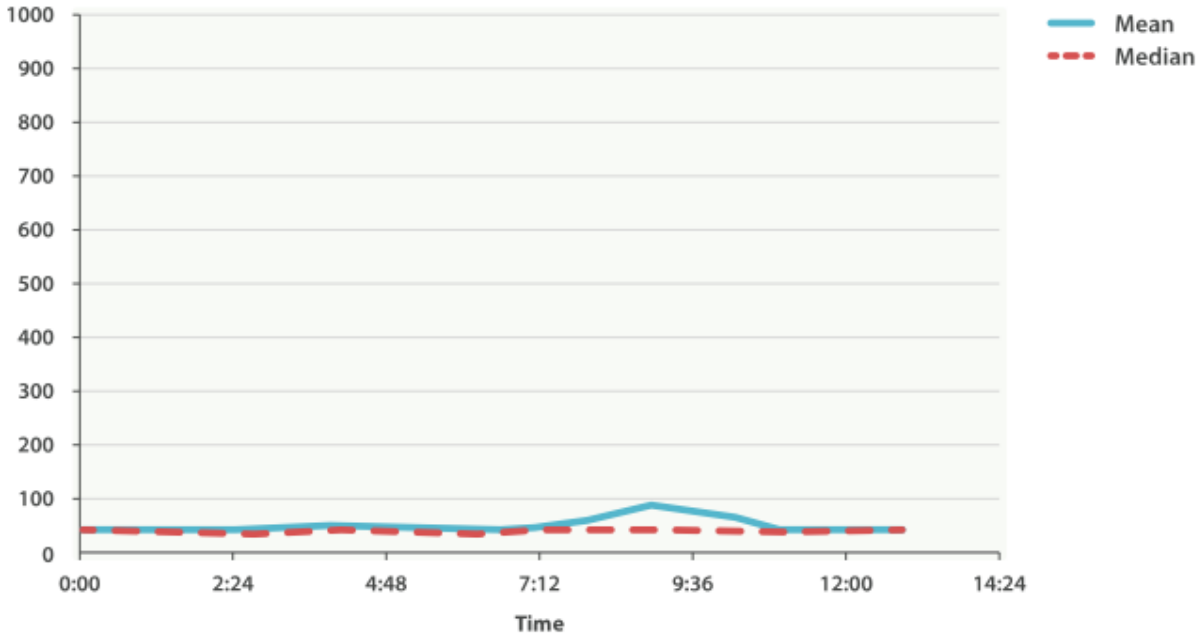

Figure 1. Average request latency over time

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 。

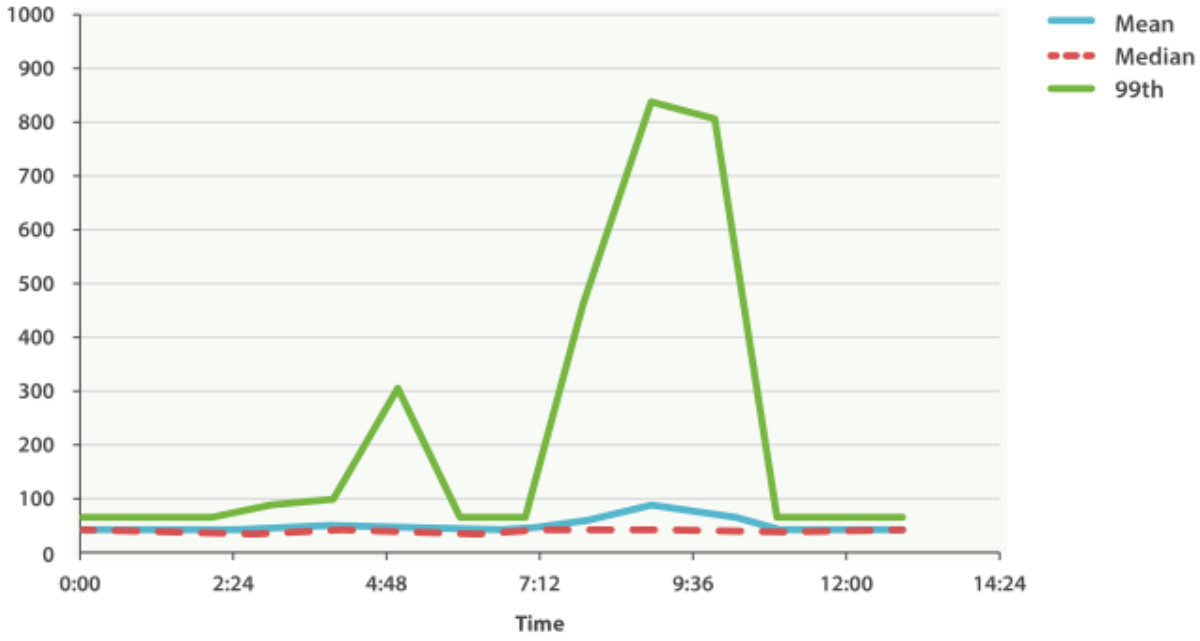

Figure 2. Average request latency with 99th percentile over time

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。