Elastic聚合

集合分类

聚合类型		描述
Metric	度量	一些数学运算，可以对文档进行统计分析
Bucket	分组桶	一些列满足特点条件的文档聚合
Pipeline	管道	对其他聚合结果，进行二次聚合
Matrix	矩阵	支持对多个文档的操作并且提供一个结果矩阵

度量聚合

- 度量聚合从文档中提c取出来值并进行计算。这些值通常从文档中的字段（使用数据字段）中提取出来，但也可以使用脚本进行计算。
- 数字型度量聚合是一种特殊类型的度量聚合，输出数字类型的值。
- 聚合输出一个数字指标（例如平均值聚合）称为单值数字型度量聚合，产生多个指标值（例如统计聚合）称为多值数字型度量聚合。
- 当这些聚合直接作为一些分组聚合的子聚合时，单值和多值数字型度量聚合的内容就会发挥巨大的作用，例如分组聚合可以对度量聚合后的返回的值进行排序。

求和 Sum

- 对聚合文档提取的数字进行求和
- 包括 **简单字段求和** 和 **通过脚本提取**
- 可以使用missing参数设置不存在的值的默认统计值

JSON | 复制代码

```
"aggs": {
    "price_sum": {
      "sum": {
        "field": "price",
        "missing": 10
      }
    },
    "price_sum_scr_field": {
      "sum": {
       "script": "doc.price.value"
      }
    },
    "price_sum_sct": {
      "sum": {
        "field": "price",
        "script": {
          "lang":   "expression",
          "source": "_value * sp",
          "params": {
             "sp": 0.5
          }
        }
      }
    }
  }
```

上面包含三种查询，

第一个 通过field获取求和字段，并且设置不存在的值为统计时为10

第二个 通过脚本获取统计参数

第三个 通过field获取求和字段，通过脚本将获取的值乘 0.5 之后进行统计

其他类似度量聚合

- 其他的一些统计的求和用法差不多

维度	关键字	
计数	value_count	
最大值	max	
最小值	min	
平均数	avg	
统计聚合	stats	一次将最大值，最小值，和，计数，平均数 返回

基数聚合

- 计算某个字段有多少个不同的值，类似于 count(DISTINCT field)
- 基于 HyperLogLog++ 实现的，所以其统计值并不一定是准确的值
- 支持脚本，不过使用脚本的时候，性能损失很大
- 通过参数 precision_threshold 可以设置准确性，使用内存来换取准确性。
  - 在大于这个值的时候结果比较模糊
  - 小于这个值的时候，接近于准确
  - 最大参数为40000，默认 3000

JSON 复制代码

```
  "aggs": {
    "price_count": {
      "cardinality": {
        "field": "price",
        "precision_threshold": 100,
        "missing": 1
      }
    }
  }
```

百分位数统计

百分位数是什么

如果将一组数据从小到大排序，并计算相应的累计百分位，则某一百分位所对应数据的值就称为这一百分位的百分位数。可表示为：一组n个观测值按数值大小排列。如，处于p%位置的值称第p百分位数。

即如果有某个百分比的数据低于一个值，这个值便叫某个百分比的百分位数。

例如：你是班上 20个学生里身高第四的学生，80% 的学生比你矮，你身高是第 80个百分位数（百分等级是 80%）。如果你的身高是 1.85m，"1.85m" 是班上身高的 80% 百分数。

如何计算

JSON | 复制代码

```
原数据：
10 12 16 20 25 30 40 45

我们需要计算 50% 的 百分位数

公式为 Location= (n+1) * percentile


L = (8+1)* 50 /100 = 4.5

此时计算出的值为4.5，不是整数，那么还需要计算 4位置 和 5位置 直接的百分位数
计算公式为：

Location = Location(L/1)+ percentile * (L/1+1 - L/1)

L 20 + 0.5 (35-20) = 22.5

此时说明，50%的数字的值都在 22.5 之下，比22.5小
```

```
    "price_count" : {
      "values" : {
        "1.0" : 10.0,
        "5.0" : 10.0,
        "25.0" : 14.0,
        "50.0" : 22.5,
        "75.0" : 35.0,
        "95.0" : 45.0,
        "99.0" : 45.0
      }
    }
```

如何使用

JSON | 复制代码

```
  "aggs": {
    "price_count": {
      "percentiles": {
        "field": "price",
        "percents": [
          1,
          5,
          25,
          50,
          75,
          95,
          99
        ]
      }
    }
  }

resp:
  "aggregations" : {
    "price_count" : {
      "values" : {
        "1.0" : 10.0,
        "5.0" : 10.0,
        "25.0" : 14.0,
        "50.0" : 22.5,
        "75.0" : 35.0,
        "95.0" : 45.0,
        "99.0" : 45.0
      }
    }
  }

```

百分数 数值聚合

- 上面的百分位统计是统计**每个百分位的值是多少**，而这个统计是统计 **当前值在文档的百分位中是多少**
- 是百分位统计的逆计算

JSON 复制代码

```
  "aggs": {
    "price_count": {
      "percentile_ranks": {
        "field": "price",
        "values": [14,22.5]
      }
    }
  }

resp:
    "price_count" : {
      "values" : {
        "14.0" : 25.0,
        "22.5" : 50.0
      }
    }
```

脚本统计

- 当es提供的统计不能满足我们的要求的时候，我们可以选择自己通过脚本实现统计

JSON 复制代码

```
  "aggs": {
    "price_count": {
      "scripted_metric": {
         "init_script": "state.transactions = []",
         "map_script": "state.transactions.add(doc.price.value)",
         "combine_script": "double profit = 0; for (t in state.transactions) { profit += t } return profit",
         "reduce_script": "double profit = 0; for (a in states) { profit += a } return profit"
      }
    }
  }
```

脚本统计分为4个阶段

1. 初始化脚本 (init_script) 在任何文档的收集之前执行。允许聚合设置任何初始化状态。
2. 映射脚本（map_script）,每个被采集的文档都会执行一次脚本。这是唯一必须的脚本。如果没有指定联合脚本，结果状态需要存储在一个名为_agg的对象中

3. 联合脚本（combine_script）,每个分片会在文档采集结束的时候执行一次脚本。允许聚合从每个分片中统一状态。如果没有提供联合脚本，联合阶段就会返回聚合变量。
4. 归纳脚本（reduce_script）。在所有分片返回结果之后，请求节点执行一次脚本。这个脚本用于访问_aggs变量，_aggs变量是每个分片执行联合脚本之后的结果数组。如果没有提供归纳脚本，归纳阶段会返回_aggs变量。

初始化前
state 被初始化成一个空对象

```
state:{}
```

init_script
初始化后，每个分片都会创建一个transactions空数组

```
state:
{
    transactions:[]
}
```

```
state:
{
    transactions:[]
}
```

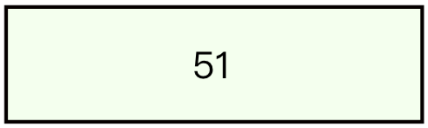
map_script
获取需要聚合的字段，填充到数组中

```
state:
{
    transactions:[10，11]
}
```

```
state:
{
    transactions:[14，15]
}
```

combine_script
在需要聚合的字段收集完成之后，每个分片进行求和，然后返回协调节点

reduce_script
协调节点获取到每个节点的数据，将数据进行集合



地理边界聚合

- 为一个geo类型的字段计算包含所有点的边界框。
- 返回左上 和 右下的点

JSON | 复制代码

```
    "aggs" : {
        "viewport" : {
            "geo_bounds" : {
                "field" : "location",
                "wrap_longitude" : true
            }
        }
    }
wrap_longitude: 用于指定是否应允许边界框与国际日期线重叠

resp:
  "aggregations" : {
    "viewport" : {
      "bounds" : {
        "top_left" : {
          "lat" : 35.234553990885615,
          "lon" : 107.80624595470726
        },
        "bottom_right" : {
          "lat" : 33.65711996331811,
          "lon" : 110.34507296048105
        }
      }
    }
  }
```

地理中心聚合

- 为一个geo类型的字段计算包含所有点的中心点

JSON | 复制代码

```json
    "aggs" : {
        "centroid" : {
            "geo_centroid" : {
                "field" : "location"
            }
        }
    }


  "aggregations" : {
    "centroid" : {
      "location" : {
        "lat" : 34.5458007780835,
        "lon" : 109.24376476183534
      },
      "count" : 5
    }
  }
```

分组聚合 桶聚合

- 分组聚合不像度量聚合那样直接通过字段聚合，而是根据字段的值，创建分组桶。
- 每个分组桶都和一个标准（取决于聚合类型）相关联，该标准确定当前上下文中的文档是否“落入”其中。换句话说，存储桶有效地定义了文档集。除了存储桶本身之外，`bucket`聚合还计算并返回“落入”每个存储桶的文档数量。
- 分组聚合还可以实现子聚合，这些子聚合可以针对父聚合的分组桶进行聚合。
- 可以定义单分组桶，多分组桶，动态创建分组桶，等
- 单个请求最多返回的分组桶受 search.max_buckets 参数影响，默认-1（不启用），但是这个时候最多返回10,000个分组

Terms Aggregation

- 按照某个字段中的值来分类，基于字段的值动态创建桶
- 字段需要打开fiel data 才能使用terms
  - keyword 需要支持doc_value
  -

JSON | 复制代码

```
  "aggs": {
    "type_terms": {
      "terms": {
        "field": "type.keyword",
        "size": 10
      }
    }
  }

resp:

  "aggregations" : {
    "type_terms" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 0,
      "buckets" : [
        {
          "key" : "t1",
          "doc_count" : 3
        },
        {
          "key" : "t4",
          "doc_count" : 3
        },
        {
          "key" : "t2",
          "doc_count" : 1
        },
        {
          "key" : "t3",
          "doc_count" : 1
        }
      ]
    }
  }

```

- doc_count_error_upper_bound: 每个索引文档的计数错误上限。
- sum_other_doc_count: 没有被放在桶中的文档计数总和

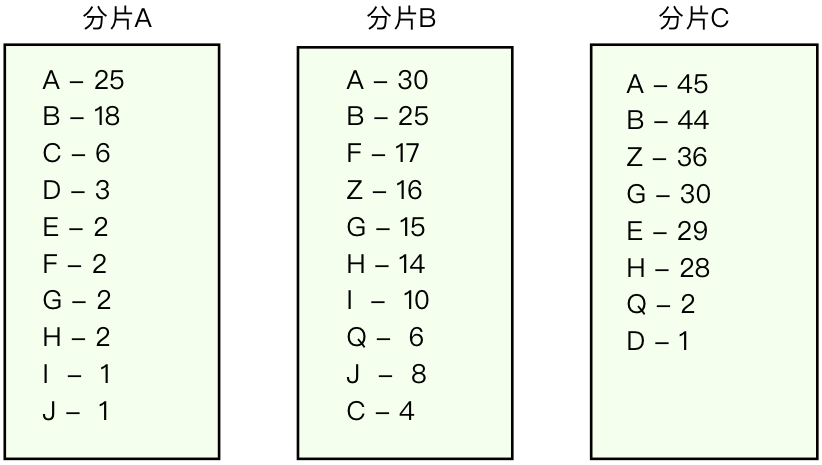
Size

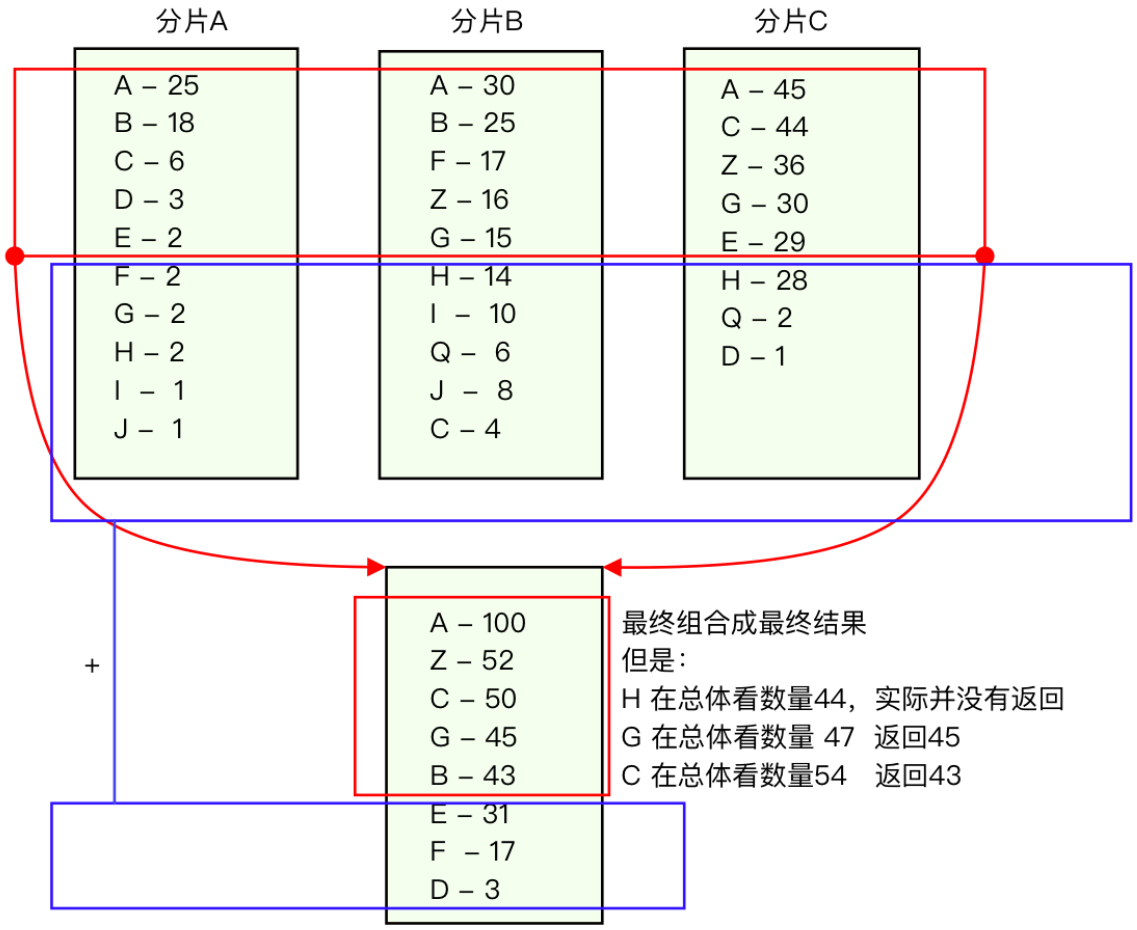
- 返回桶的数量

- 如果分组多于size，则返回的可能不是所有的数据，而且返回的可能不是前几个桶，有可能有偏差，数量也有可能有偏差
- 如果需要对聚合的结果进行分页，则应该使用composite 聚合，而不是把size设置为一个很大的值

文件数数一个大概值

- 因为在聚合中，每个分片都提供了自己的视图，将他们组合在一起形成真正的视图





无论在单个组的统计，还是总体组的返回
在如果不是全量返回的时候，都有问题

聚合的doc_count_error_upper_bound：
此聚合的错误值的46，说明在没有展示的聚合中，最坏的的可能值是46，我们可以知道，下一个H的值是44，说明这个值就是大于等于未展示的。

每个桶的doc_count_error_upper_bound：
表示这个桶的计数最多缺少多少个，例如C，在分片A 分片C返回值了，在分片B没有返回那么，那么取分片B返回数据的最小值，就肯定是在分片B中有可能缺少的最大值。此时分片B返回的数据最小为15，就说明最多缺少15个统计，但是我们可以看到，实际上，只是缺少4个，这个值表示的就最坏的情况下的错误值。大于等于未展示的。

在返回值的参数就是描述这个

- doc_count_error_upper_bound
  - 表示没有在本次返回，但是可能存在的潜在聚合结果。被遗漏的可能的最大值。
  - 主聚合的值取的是每个分片取的数据的最小值相加
  - 各个桶取的是 通过将所有未返回该词条的分片返回的最后一个词的文档计数相加得出

的，表示文档计数中最坏的情况。表示最坏少多少个元素

- sum_other_doc_count：表示的是蓝框当中的部分，本次返回之外的聚合文档数
- 从代码上看，我们可以看到如果我们这个值应该取的是每个桶的最小的一个值累加

JSON 复制代码

```
从源码看的话：

    private long getDocCountError(A terms) {
        int size = terms.getBuckets().size();
        if (size == 0 || size < terms.getShardSize() ||
isKeyOrder(terms.getOrder())) {
            return 0;
        } else if (InternalOrder.isCountDesc(terms.getOrder())) {
            if (terms.getDocCountError() > 0) {
                // If there is an existing docCountError for this agg
then
                // use this as the error for this aggregation
                return terms.getDocCountError();
            } else {
                // otherwise use the doc count of the last term in the
                // aggregation
                return
terms.getBuckets().stream().mapToLong(AbstractTermsBucket::getDocCount).m
in().getAsLong();
            }
        } else {
            return -1;
        }
    }

```

shard_size

- shard_size参数就是哪个红框返回分组数据量
- 默认的分组数为 `size` `(size * 1.5 + 10)`
- 当shard_size值越大的时候，返回的数据越准确，当上图我们把shard_size设置为8的时候，H也可以被统计进去，结果就不一样了。但是这样也会带来计算成本的增加
- shard_size不能小于size，因为没有意义，即使设置，es也会重置为size

展示每个桶的错误

- show_term_doc_count_error 设置为true的时候，可以展示每个桶的可能错误值
- 各个桶取的是 通过将所有未返回该词条的分片返回的最后一个词的文档计数相加得出的，表

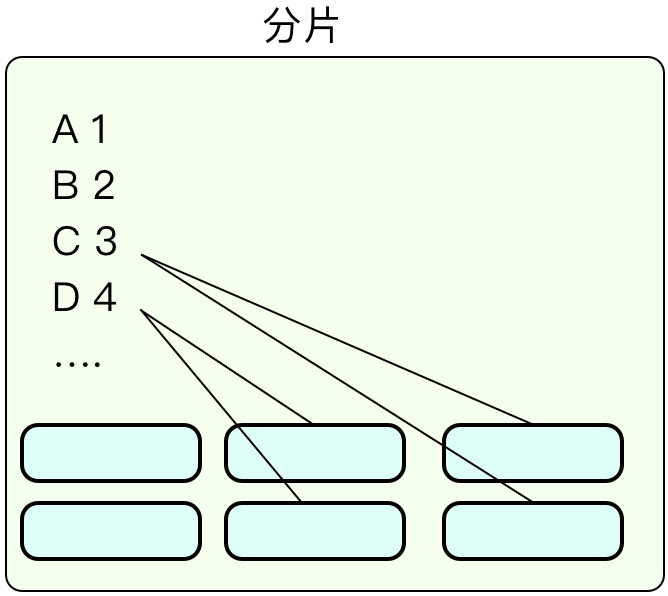
示文档计数中最坏的情况。表示最坏少多少个元素

- 代码在 org.elasticsearch.search.aggregations.bucket.terms.AbstractInternalTerms#reduce

Execution hint

- `map`，直接使用该字段的字符串值来做聚合
- `global_ordinals`：默认选项，

global_ordinals



- Global Ordinals 一个分片级别全局的映射字典，即用一个序号来代表一个唯一字符串，在进行terms统计的时候，直接为每一个序号生成一个桶，在获取文档的时候，就不用把字段值加载到内存中进行计算，只要加载字典的序号就可以，然后遍历映射到桶中，就可以很快的统计出每个桶的文档数
- 使用Global Ordinals的目的在于减少内存使用，加快聚合统计，这种计算方式主要开销在构建global ordinals和分配bucket上，如果索引包含的原始文档非常多，查询结果包含的文档也很多，那么默认的这种计算方式是内存消耗最小，速度最快的。
- **问题**：
  - 这种方式是懒加载的，而且是在开始之前进行的，这样导致在字段种类特别大的情况下，即使过滤的文档很少，也会很慢
  - 关系在shard被触发refresh以后就会失效。下次使用的话就需要重新构建，
  - 而且这份数据常驻内存
- **优化方法：**
  - 增加分片数量，即减少每个分片创建映射的时间
  - 延长refresh_interval的时间，但是数据的实时性就会打折扣

- 修改execution_hint的值：当把值从 `global_ordinals` 改为 `map` ，每次聚合的时候直接把值加载到内存中计算，减去了构建字典的耗时。当查询的结果集很小的情况下，可以使用map的模式不去构建字典。使用map还是global_ordinals的取决于构建字典的开销与加载原始字典的开销。当结果集大到一定程序，map的内存开销的代价可能抵消了构建字典的开销。

收集模式

- depth_first 深度优先搜索，直接进行子聚合的计算
- breadth_first 广度优先搜索，先计算出当前聚合的结果，针对这个结果在对子聚合进行计算
  - 会把上层的桶的文档集合存储起来，以供后续使用，会产生内存开销
  - 该内存开销与匹配文档的数量成线性关系

参数Order

- 指定了最后返回结果的排序方式，默认是按照doc_count降序排列。一般不建议按照升序排序，因为这样会增加文档计数的错误。

JSON 复制代码

```
1. 升序排行
"aggs" : {
        "genres" : {
            "terms" : {
                "field" : "genre",
                "order" : { "_count" : "asc" }
            }
        }
}
2. 按照key排序
    "aggs" : {
        "genres" : {
            "terms" : {
                "field" : "genre",
                "order" : { "_key" : "asc" }
            }
        }
    }
3.
```

min_doc_count

- 使用min_doc_count参数的时候，只会返回大于数值的的匹配分组桶

include exclude

- 包含 和 排除 的分组key

```JSON
  "aggs": {
    "heshen_terms": {
      "terms": {
        "field": "type",
        "size": 5,
        "shard_size": 5,
        "show_term_doc_count_error": true,
        "include": ["Product A","Product B"],
        "exclude": ["Product A"]
      }
    }
  }
```

Missing value

- 文档缺少值的时候的处理

管道聚合

概念： 支持对聚合分析的结果，再一次进行聚合分析

管道聚合的结果会添加到返回当中，根据位置不同分为两类：

Sibling 同级

包括：max_bucket,min_bucket ,avg_bucket ,sum_bucket , stats_bucket , percentiles_bucket

Parent 父级

包括：derivative （求导） ， cumulative_sum （累计求和） ， moving_fn（滑动窗口）