主讲老师：Fox

> 有道笔记地址： https://note.youdao.com/s/U3mccx9s

1. ES分词器详解

1.1 基本概念

分词器官方称之为文本分析器，顾名思义，是对文本进行分析处理的一种手段，基本处理逻辑为按照预先制定的分词规则，把原始文档分割成若干更小粒度的词项，粒度大小取决于分词器规则。

```
#ES的默认分词设置是standard，会单字拆分
POST _analyze
{
  "analyzer":"standard",
  "text":"中华人民共和国"
}

#ik_smart:会做最粗粒度的拆
POST _analyze
{
  "analyzer": "ik_smart",
  "text": "中华人民共和国"
 }

#ik_max_word:会将文本做最细粒度的拆分
POST _analyze
{
  "analyzer":"ik_max_word",
  "text":"中华人民共和国"
}
```

```
{
  "tokens" : [
    {
      "token" : "中华人民共和国",
      "start_offset" : 0,
      "end_offset" : 7,
      "type" : "CN_WORD",
      "position" : 0
    },
    {
      "token" : "中华人民",
      "start_offset" : 0,
      "end_offset" : 4,
      "type" : "CN_WORD",
      "position" : 1
    },
    {
      "token" : "中华",
      "start_offset" : 0,
      "end_offset" : 2,
      "type" : "CN_WORD",
      "position" : 2
    },
    {
```

1.2 分词发生时期

分词器的处理过程发生在 Index Time 和 Search Time 两个时期。

- Index Time：文档写入并创建倒排索引时期，其分词逻辑取决于映射参数analyzer。
- Search Time：搜索发生时期，其分词仅对搜索词产生作用。

1.3 分词器的组成

- 切词器（Tokenizer）：用于定义切词（分词）逻辑
- 词项过滤器（Token Filter）：用于对分词之后的单个词项的处理逻辑
- 字符过滤器（Character Filter）：用于处理单个字符

注意：

- 分词器不会对源数据造成任何影响，分词仅仅是对倒排索引或者搜索词的行为。

切词器：Tokenizer

tokenizer 是分词器的核心组成部分之一，其主要作用是分词，或称之为切词。主要用来对原始文本进行细粒度拆分。拆分之后的每一个部分称之为一个 Term，或称之为一个词项。可以把切词器理解为预定义的切词规则。官方内置了很多种切词器，默认的切词器位 standard。

词项过滤器：Token Filter

词项过滤器用来处理切词完成之后的词项，例如把大小写转换，删除停用词或同义词处理等。官方同样预置了很多词项过滤器，基本可以满足日常开发的需要。当然也是支持第三方也自行开发的。

```
GET _analyze
{
  "filter" : ["lowercase"],
  "text" : "WWW ELASTIC ORG CN"
}

GET _analyze
{
  "tokenizer" : "standard",
  "filter" : ["uppercase"],
  "text" : ["www.elastic.org.cn","www elastic org cn"]
}
```

停用词

在切词完成之后，会被干掉词项，即停用词。停用词可以自定义

英文停用词（**english**）：a, an, and, are, as, at, be, but, by, for, if, in, into, is, it, no, not, of, on, or, such, that, the, their, then, there, these, they, this, to, was, will, with。

中日韩停用词（**cjk**）：a, and, are, as, at, be, but, by, for, if, in, into, is, it, no, not, of, on, or, s, such, t, that, the, their, then, there, these, they, this, to, was, will, with, www。

```
GET _analyze
{
  "tokenizer": "standard",
  "filter": ["stop"],
  "text": ["What are you doing"]
}

### 自定义 filter
```

```
DELETE test_token_filter_stop
PUT test_token_filter_stop
{
  "settings": {
    "analysis": {
      "filter": {
        "my_filter": {
          "type": "stop",
          "stopwords": [
            "www"
          ],
          "ignore_case": true
        }
      }
    }
  }
}
GET test_token_filter_stop/_analyze
{
  "tokenizer": "standard",
  "filter": ["my_filter"],
  "text": ["What www WWW are you doing"]
}
```

同义词

同义词定义规则

- a, b, c => d：这种方式，a、b、c 会被 d 代替。
- a, b, c, d：这种方式下，a、b、c、d 是等价的。

```
PUT test_token_filter_synonym
{
  "settings": {
    "analysis": {
      "filter": {
        "my_synonym": {
          "type": "synonym",
          "synonyms": [ "good, nice => excellent" ] //good, nice, excellent
```

```
        }
      }
    }
  }
}
GET test_token_filter_synonym/_analyze
{
  "tokenizer": "standard",
  "filter": ["my_synonym"],
  "text": ["good"]
}
```

字符过滤器：Character Filter

分词之前的预处理，过滤无用字符。

```
PUT <index_name>
{
  "settings": {
    "analysis": {
      "char_filter": {
        "my_char_filter": {
          "type": "<char_filter_type>"
        }
      }
    }
  }
}
```

type：使用的字符过滤器类型名称，可配置以下值：

- html_strip
- mapping
- pattern_replace

HTML 标签过滤器：HTML Strip Character Filter

字符过滤器会去除 HTML 标签和转义 HTML 元素，如 、&

```
PUT test_html_strip_filter
{
  "settings": {
    "analysis": {
      "char_filter": {
        "my_char_filter": {
          "type": "html_strip",  // html_strip 代表使用 HTML 标签过滤器
          "escaped_tags": [      // 当前仅保留 a 标签
            "a"
          ]
        }
      }
    }
  }
}
GET test_html_strip_filter/_analyze
{
  "tokenizer": "standard",
  "char_filter": ["my_char_filter"],
  "text": ["<p>I&apos;m so <a>happy</a>!</p>"]
}
```

参数：escaped_tags：需要保留的 html 标签

字符映射过滤器：Mapping Character Filter

通过定义映替换为规则，把特定字符替换为指定字符

```
PUT test_html_strip_filter
{
  "settings": {
    "analysis": {
      "char_filter": {
        "my_char_filter": {
          "type": "mapping", // mapping 代表使用字符映射过滤器
          "mappings": [                          // 数组中规定的字符会被等价替换为 => 指定的字符
            "滚 => *",
            "垃 => *",
            "圾 => *"
```

```
          ]
        }
      }
    }
  }
}
GET test_html_strip_filter/_analyze
{
  //"tokenizer": "standard",
  "char_filter": ["my_char_filter"],
  "text": "你就是个垃圾！滚"
}
```

正则替换过滤器：Pattern Replace Character Filter

```
PUT text_pattern_replace_filter
{
  "settings": {
    "analysis": {
      "char_filter": {
        "my_char_filter": {
          "type": "pattern_replace", // pattern_replace 代表使用正则替换过滤器

          "pattern": """(\d{3})\d{4}(\d{4})""", // 正则表达式
          "replacement": "$1****$2"
        }
      }
    }
  }
}
GET text_pattern_replace_filter/_analyze
{
  "char_filter": ["my_char_filter"],
  "text": "您的手机号是18868686688"
}
```

1.4 倒排索引的数据结构

当数据写入 ES 时，数据将会通过 分词 被切分为不同的 term，ES 将 term 与其对应的文档列表建立一种映射关系，这种结构就是 倒排索引。如下图所示：

为了进一步提升索引的效率，ES 在 term 的基础上利用 term 的前缀或者后缀构建了 term index, 用于对 term 本身进行索引，ES 实际的索引结构如下图所示：

这样当我们去搜索某个关键词时，ES 首先根据它的前缀或者后缀迅速缩小关键词的在 term dictionary 中的范围，大大减少了磁盘IO的次数。

- 单词词典（Term Dictionary）：记录所有文档的单词，记录单词到倒排列表的关联关系
  - 常用字典数据结构：https://www.cnblogs.com/LBSer/p/4119841.html
- 倒排列表(Posting List)-记录了单词对应的文档结合，由倒排索引项组成
- 倒排索引项(Posting)：
  - 文档ID
  - 词频TF–该单词在文档中出现的次数，用于相关性评分
  - 位置(Position)-单词在文档中分词的位置。用于短语搜索（match phrase query)
  - 偏移(Offset)-记录单词的开始结束位置，实现高亮显示

Elasticsearch 的JSON文档中的每个字段，都有自己的倒排索引。

可以指定对某些字段不做索引：

- 优点：节省存储空间
- 缺点: 字段无法被搜索

2. 相关性详解

搜索是用户和搜索引擎的对话，用户关心的是搜索结果的相关性

- 是否可以找到所有相关的内容
- 有多少不相关的内容被返回了
- 文档的打分是否合理
- 结合业务需求，平衡结果排名

2.1 什么是相关性（Relevance）

搜索的相关性算分，描述了一个文档和查询语句匹配的程度。ES 会对每个匹配查询条件的结果进行算分_score。打分的本质是排序，需要把最符合用户需求的文档排在前面。

如下例子：显而易见，查询JAVA多线程设计模式，文档id为2,3的文档的算分更高

关键词	文档ID

JAVA	1,2,3
设计模式	1,2,3,4,5,6
多线程	2,3,7,9

如何衡量相关性：

- Precision(查准率)-尽可能返回较少的无关文档
- Recall(查全率)–尽量返回较多的相关文档
- Ranking -是否能够按照相关度进行排序

2.2 相关性算法

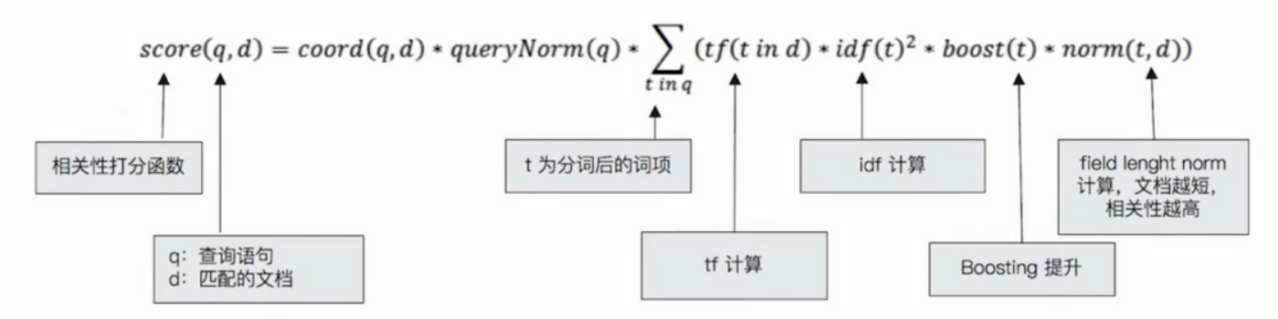
ES 5之前，默认的相关性算分采用TF-IDF，现在采用BM 25。

TF-IDF

TF-IDF（term frequency–inverse document frequency）是一种用于信息检索与数据挖掘的常用加权技术。

- TF-IDF被公认为是信息检索领域最重要的发明，除了在信息检索，在文献分类和其他相关领域有着非常广泛的应用。
- IDF的概念，最早是剑桥大学的“斯巴克.琼斯”提出
  - 1972年——“关键词特殊性的统计解释和它在文献检索中的应用”，但是没有从理论上解释IDF应该是用log(全部文档数/检索词出现过的文档总数)，而不是其他函数，也没有做进一步的研究
  - 1970，1980年代萨尔顿和罗宾逊，进行了进一步的证明和研究，并用香农信息论做了证明http://www.staff.city.ac.uk/~sb317/papers/foundations_bm25_review.pdf
- 现代搜索引擎，对TF-IDF进行了大量细微的优化

Lucene中的TF-IDF评分公式：



- **TF是词频(Term Frequency)**

检索词在文档中出现的频率越高，相关性也越高。

```
词频（TF） = 某个词在文档中出现的次数 / 文档的总词数
```

- **IDF是逆向文本频率(Inverse Document Frequency)**

每个检索词在索引中出现的频率，频率越高，相关性越低。总文档中有些词比如“是”、“的”、“在” 在所有文档中出现频率都很高，并不重要，可以减少多个文档中都频繁出现的词的权重。

```
逆向文本频率（IDF）= log（语料库的文档总数 / （包含该词的文档数+1））
```

- **字段长度归一值（field-length norm)**

检索词出现在一个内容短的 title 要比同样的词出现在一个内容长的 content 字段权重更大。

以上三个因素——词频（term frequency）、逆向文本频率（inverse document frequency）和字段长度归一值（field-length norm）——是在索引时计算并存储的，最后将它们结合在一起计算单个词在特定文档中的权重。

BM25

BM25 就是对 TF-IDF 算法的改进，对于 TF-IDF 算法，TF(t) 部分的值越大，整个公式返回的值就会越大。BM25 就针对这点进行来优化，随着TF(t) 的逐步加大，该算法的返回值会趋于一个数值。

- 从ES 5开始，默认算法改为BM 25
- 和经典的TF-IDF相比,当TF无限增加时，BM 25算分会趋于一个数值

- BM 25的公式

$$\mathrm{bm25}(d) = \sum_{t \in q, f_{t,d} > 0} \log\left(1 + \frac{N - df_t + 0.5}{df_t + 0.5}\right) \cdot \frac{f_{t,d}}{f_{t,d} + k \cdot (1 - b + b\frac{\mathrm{l}(d)}{avgdl})}$$

2.3 通过Explain API查看TF-IDF

示例：

```
PUT /test_score/_bulk
{"index":{"_id":1}}
{"content":"we use Elasticsearch to power the search"}
{"index":{"_id":2}}
{"content":"we like elasticsearch"}
{"index":{"_id":3}}
{"content":"Thre scoring of documents is caculated by the scoring formula"}
{"index":{"_id":4}}
{"content":"you know,for search"}

GET /test_score/_search
{
  "explain": true,
  "query": {
    "match": {
      "content": "elasticsearch"
    }
  }
}

GET /test_score/_explain/2
{
  "explain": true,
  "query": {
    "match": {
      "content": "elasticsearch"
    }
  }
}
```

2.4 Boosting Query

Boosting是控制相关度的一种手段。可以通过指定字段的boost值影响查询结果

参数boost的含义：

- 当boost > 1时，打分的权重相对性提升
- 当0 < boost <1时，打分的权重相对性降低
- 当boost <0时，贡献负分

应用场景：希望包含了某项内容的结果不是不出现，而是排序靠后。

```
POST /blogs/_bulk
{"index":{"_id":1}}
{"title":"Apple iPad","content":"Apple iPad,Apple iPad"}
{"index":{"_id":2}}
{"title":"Apple iPad,Apple iPad","content":"Apple iPad"}

GET /blogs/_search
{
  "query": {
    "bool": {
      "should": [
        {
          "match": {
            "title": {
              "query": "apple,ipad",
              "boost": 1
            }
          }
        },
        {
          "match": {
            "content": {
              "query": "apple,ipad",
              "boost": 4
            }
          }
        }
      ]
    }
  }
}
```

案例：要求苹果公司的产品信息优先展示

```
POST /news/_bulk
{"index":{"_id":1}}
{"content":"Apple Mac"}
{"index":{"_id":2}}
{"content":"Apple iPad"}
{"index":{"_id":3}}
{"content":"Apple employee like Apple Pie and Apple Juice"}


GET /news/_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "content": "apple"
        }
      }
    }
  }
}

```

利用must not排除不是苹果公司产品的文档

```
GET /news/_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "content": "apple"
        }
```

```
      },
      "must_not": {
        "match":{
          "content": "pie"
        }
      }
    }
  }
}
```

利用negative_boost降低相关性

对某些返回结果不满意，但又不想排除掉（ must_not)，可以考虑boosting query的negative_boost。

- negative_boost 对 negative部分query生效
- 计算评分时,boosting部分评分不修改，negative部分query乘以negative_boost值
- negative_boost取值:0-1.0，举例:0.3

```
GET /news/_search
{
  "query": {
    "boosting": {
      "positive": {
        "match": {
          "content": "apple"
        }
      },
      "negative": {
        "match": {
          "content": "pie"
        }
      },
      "negative_boost": 0.2
    }
  }
}
```

3. 单字符串多字段查询

三种场景：

- **最佳字段(Best Fields)**

当字段之间相互竞争，又相互关联。例如，对于博客的 title和 body这样的字段，评分来自最匹配字段

- **多数字段(Most Fields)**

处理英文内容时的一种常见的手段是，在主字段(English Analyzer)，抽取词干，加入同义词，以匹配更多的文档。相同的文本，加入子字段（Standard Analyzer），以提供更加精确的匹配。其他字段作为匹配文档提高相关度的信号，匹配字段越多则越好。

- **混合字段(Cross Field)**

对于某些实体，例如人名，地址，图书信息。需要在多个字段中确定信息，单个字段只能作为整体的一部分。希望在任何这些列出的字段中找到尽可能多的词。

3.1 最佳字段查询Dis Max Query

将任何与任一查询匹配的文档作为结果返回，采用字段上最匹配的评分最终评分返回。

官方文档：https://www.elastic.co/guide/en/elasticsearch/reference/7.17/query-dsl-dis-max-query.html

测试

```
DELETE /blogs
PUT /blogs/_doc/1
{
    "title": "Quick brown rabbits",
    "body":  "Brown rabbits are commonly seen."
}

PUT /blogs/_doc/2
{
    "title": "Keeping pets healthy",
    "body":  "My quick brown fox eats rabbits on a regular basis."
}

```

```
POST /blogs/_search
{
    "query": {
        "bool": {
            "should": [
                { "match": { "title": "Brown fox" }},
                { "match": { "body":  "Brown fox" }}
            ]
        }
    }
}
```

思考：查询结果不符合预期，为什么?

```
PUT /blogs/_doc/1
{
  "title": "Quick brown rabbits",
  "body":  "Brown rabbits are commonly seen."
}

PUT /blogs/_doc/2
{
  "title": "Keeping pets healthy",
  "body":  "My quick brown fox eats rabbits on a regular basis."
}

POST /blogs/_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "Brown fox" }},
        { "match": { "body":  "Brown fox" }}
      ]
    }
  }
}
```

```
    "relation" : "eq"
  },
  "max_score" : 0.90425634,
  "hits" : [
    {
      "_index" : "blogs",
      "_type" : "_doc",
      "_id" : "1",
      "_score" : 0.90425634,
      "_source" : {
        "title" : "Quick brown rabbits",
        "body" : "Brown rabbits are commonly seen."
      }
    },
    {
      "_index" : "blogs",
      "_type" : "_doc",
      "_id" : "2",
      "_score" : 0.77041256,
      "_source" : {
        "title" : "Keeping pets healthy",
        "body" : "My quick brown fox eats rabbits on a regular basis."
      }
    }
  ]
```

bool should的算法过程：

- 查询should语句中的两个查询
- 加和两个查询的评分
- 乘以匹配语句的总数
- 除以所有语句的总数

上述例子中，title和body属于竞争关系，不应该将分数简单叠加，而是应该找到单个最佳匹配的字段的评分。

使用最佳字段查询dis max query

```
POST /blogs/_search
{
    "query": {
        "dis_max": {
            "queries": [
```

```
                { "match": { "title": "Brown fox" }},
                { "match": { "body":  "Brown fox" }}
            ]
        }
    }
}
```

可以通过tie_breaker参数调整

Tier Breaker是一个介于0-1之间的浮点数。0代表使用最佳匹配;1代表所有语句同等重要。

1. 获得最佳匹配语句的评分_score 。
2. 将其他匹配语句的评分与tie_breaker相乘
3. 对以上评分求和并规范化

```
POST /blogs/_search
{
    "query": {
        "dis_max": {
            "queries": [
                { "match": { "title": "Quick pets" }},
                { "match": { "body":  "Quick pets" }}
            ]
        }
    }
}


POST /blogs/_search
{
    "query": {
        "dis_max": {
            "queries": [
                { "match": { "title": "Quick pets" }},
                { "match": { "body":  "Quick pets" }}
            ],
            "tie_breaker": 0.1
        }
    }
```

```
}
```

3.2 Multi Match Query

最佳字段(Best Fields)搜索

best_fields策略获取最佳匹配字段的得分, final_score = max(其他匹配字段得分， 最佳匹配字段得分)

采用 best_fields 查询，并添加参数 tie_breaker=0.1，final_score = 其他匹配字段得分 * 0.1 + 最佳匹配字段得分

Best Fields是默认类型，可以不用指定，等价于dis_max查询方式

```
POST /blogs/_search
{
  "query": {
    "multi_match": {
      "type": "best_fields",
      "query": "Brown fox",
      "fields": ["title","body"],
      "tie_breaker": 0.2
    }
  }
}
```

案例

```
PUT /employee
{
    "settings" : {
        "index" : {
            "analysis.analyzer.default.type": "ik_max_word"
        }
    }
}

POST /employee/_bulk
```

```
{"index":{"_id":1}}
{"empId":"1","name":"员工001","age":20,"sex":"男","mobile":"19000001111","salary":23343,"deptName":"技术部","address":"湖北省武汉市洪山区光谷大厦","content":"i like to write best elasticsearch article"}
{"index":{"_id":2}}
{"empId":"2","name":"员工002","age":25,"sex":"男","mobile":"19000002222","salary":15963,"deptName":"销售部","address":"湖北省武汉市江汉路","content":"i think java is the best programming language"}
{"index":{"_id":3}}
{"empId":"3","name":"员工003","age":30,"sex":"男","mobile":"19000003333","salary":20000,"deptName":"技术部","address":"湖北省武汉市经济开发区","content":"i am only an elasticsearch beginner"}
{"index":{"_id":4}}
{"empId":"4","name":"员工004","age":20,"sex":"女","mobile":"19000004444","salary":15600,"deptName":"销售部","address":"湖北省武汉市沌口开发区","content":"elasticsearch and hadoop are all very good solution, i am a beginner"}
{"index":{"_id":5}}
{"empId":"5","name":"员工005","age":20,"sex":"男","mobile":"19000005555","salary":19665,"deptName":"测试部","address":"湖北省武汉市东湖隧道","content":"spark is best big data solution based on scala, an programming language similar to java"}
{"index":{"_id":6}}
{"empId":"6","name":"员工006","age":30,"sex":"女","mobile":"19000006666","salary":30000,"deptName":"技术部","address":"湖北省武汉市江汉路","content":"i like java developer"}
{"index":{"_id":7}}
{"empId":"7","name":"员工007","age":60,"sex":"女","mobile":"19000007777","salary":52130,"deptName":"测试部","address":"湖北省黄冈市边城区","content":"i like elasticsearch developer"}
{"index":{"_id":8}}
{"empId":"8","name":"员工008","age":19,"sex":"女","mobile":"19000008888","salary":60000,"deptName":"技术部","address":"湖北省武汉市江汉大学","content":"i like spark language"}
{"index":{"_id":9}}
{"empId":"9","name":"员工009","age":40,"sex":"男","mobile":"19000009999","salary":23000,"deptName":"销售部","address":"河南省郑州市郑州大学","content":"i like java developer"}
{"index":{"_id":10}}
{"empId":"10","name":"张湖北","age":35,"sex":"男","mobile":"19000001010","salary":18000,"deptName":"测试部","address":"湖北省武汉市东湖高新","content":"i like java developer, i also like elasticsearch"}
{"index":{"_id":11}}
{"empId":"11","name":"王河南","age":61,"sex":"男","mobile":"19000001011","salary":10000,"deptName":"销售
```

```
部","address":"河南省开封市河南大学","content":"i am not like java"}
{"index":{"_id":12}}
{"empId":"12","name":"张大学","age":26,"sex":"女","mobile":"19000001012","salary":11321,"deptName":"测试部","address":"河南省开封市河南大学","content":"i am java developer, java is good"}
{"index":{"_id":13}}
{"empId":"13","name":"李江汉","age":36,"sex":"男","mobile":"19000001013","salary":11215,"deptName":"销售部","address":"河南省郑州市二七区","content":"i like java and java is very best, i like it, do you like java"}
{"index":{"_id":14}}
{"empId":"14","name":"王技术","age":45,"sex":"女","mobile":"19000001014","salary":16222,"deptName":"测试部","address":"河南省郑州市金水区","content":"i like c++"}
{"index":{"_id":15}}
{"empId":"15","name":"张测试","age":18,"sex":"男","mobile":"19000001015","salary":20000,"deptName":"技术部","address":"河南省郑州市高新开发区","content":"i think spark is good"}


GET /employee/_search
{
  "query": {
    "multi_match": {
      "query": "elasticsearch beginner 湖北省 开封市",
      "type": "best_fields",
      "fields": [
        "content",
        "address"
      ]
    }
  },
  "size": 15
}


# 查看执行计划
GET /employee/_explain/3
{

  "query": {
    "multi_match": {
```

```
      "query": "elasticsearch beginner 湖北省 开封市",
      "type": "best_fields",
      "fields": [
        "content",
        "address"
      ]
    }
  }
}

GET /employee/_explain/3
{

  "query": {
    "multi_match": {
      "query": "elasticsearch beginner 湖北省 开封市",
      "type": "best_fields",
      "fields": [
        "content",
        "address"
      ],
      "tie_breaker": 0.1
    }
  }
}

```

使用多数字段（Most Fields）搜索

most_fields策略获取全部匹配字段的累计得分（综合全部匹配字段的得分），等价于bool should查询方式

```
GET /employee/_explain/3
{

  "query": {
```

```
    "multi_match": {
      "query": "elasticsearch beginner 湖北省 开封市",
      "type": "most_fields",
      "fields": [
        "content",
        "address"
      ]
    }
  }
}
```

案例

```
DELETE /titles
PUT /titles
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "analyzer": "english",
        "fields": {
          "std": {
            "type": "text",
            "analyzer": "standard"
          }
        }
      }
    }
  }
}

POST titles/_bulk
{ "index": { "_id": 1 }}
{ "title": "My dog barks" }
{ "index": { "_id": 2 }}
{ "title": "I see a lot of barking dogs on the road " }

```

```
# 结果与预期不匹配
GET /titles/_search
{
  "query": {
    "match": {
      "title": "barking dogs"
    }
  }
}
```

用广度匹配字段title包括尽可能多的文档——以提升召回率——同时又使用字段title.std 作为信号将相关度更高的文档置于结果顶部。

```
GET /titles/_search
{
  "query": {
    "multi_match": {
      "query": "barking dogs",
      "type": "most_fields",
      "fields": [
        "title",
        "title.std"
      ]
    }
  }
}
```

每个字段对于最终评分的贡献可以通过自定义值boost 来控制。比如，使title 字段更为重要,这样同时也降低了其他信号字段的作用：

```
#增加title的权重
GET /titles/_search
{
  "query": {
    "multi_match": {
      "query": "barking dogs",
      "type": "most_fields",
```

```
      "fields": [
        "title^10",
        "title.std"
      ]
    }
  }
}
```

3.3 跨字段（Cross Field）搜索

搜索内容在多个字段中都显示，类似bool+dis_max组合

```
DELETE /address
PUT /address
{
    "settings" : {
        "index" : {
            "analysis.analyzer.default.type": "ik_max_word"
        }
    }
}

PUT /address/_bulk
{ "index": { "_id": "1"} }
{"province": "湖南","city": "长沙"}
{ "index": { "_id": "2"} }
{"province": "湖南","city": "常德"}
{ "index": { "_id": "3"} }
{"province": "广东","city": "广州"}
{ "index": { "_id": "4"} }
{"province": "湖南","city": "邵阳"}

#使用most_fields的方式结果不符合预期，不支持operator
GET /address/_search
{
  "query": {
    "multi_match": {
      "query": "湖南常德",
```

```
      "type": "most_fields",
      "fields": ["province","city"]
    }
  }
}

# 可以使用cross_fields，支持operator
#与copy_to相比，其中一个优势就是它可以在搜索时为单个字段提升权重。
GET /address/_search
{
  "query": {
    "multi_match": {
      "query": "湖南常德",
      "type": "cross_fields",
      "operator": "and",
      "fields": ["province","city"]
    }
  }
}
```

可以用copy...to 解决，但是需要额外的存储空间

```
DELETE /address
# copy_to参数允许将多个字段的值复制到组字段中，然后可以将其作为单个字段进行查询
PUT /address
{
  "mappings" : {
      "properties" : {
        "province" : {
          "type" : "keyword",
          "copy_to": "full_address"
        },
        "city" : {
          "type" : "text",
          "copy_to": "full_address"
        }
      }
    },
```

```
    "settings" : {
        "index" : {
            "analysis.analyzer.default.type": "ik_max_word"
        }
    }
}

PUT /address/_bulk
{ "index": { "_id": "1"} }
{"province": "湖南","city": "长沙"}
{ "index": { "_id": "2"} }
{"province": "湖南","city": "常德"}
{ "index": { "_id": "3"} }
{"province": "广东","city": "广州"}
{ "index": { "_id": "4"} }
{"province": "湖南","city": "邵阳"}

GET /address/_search
{
  "query": {
    "match": {
      "full_address": {
        "query": "湖南常德",
        "operator": "and"
      }
    }
  }
}

```

4. ElasticSearch聚合操作

Elasticsearch除搜索以外，提供了针对ES 数据进行统计分析的功能。聚合(aggregations)可以让我们极其方便的实现对数据的统计、分析、运算。例如：

- 什么品牌的手机最受欢迎?
- 这些手机的平均价格、最高价格、最低价格?

- 这些手机每月的销售情况如何?

使用场景

聚合查询可以用于各种场景，比如商业智能、数据挖掘、日志分析等等。

- 电商平台的销售分析：统计每个地区的销售额、每个用户的消费总额、每个产品的销售量等，以便更好地了解销售情况和趋势。
- 社交媒体的用户行为分析：统计每个用户的发布次数、转发次数、评论次数等，以便更好地了解用户行为和趋势，同时可以将数据按照地区、时间、话题等维度进行分析。
- 物流企业的运输分析：统计每个区域的运输量、每个车辆的运输次数、每个司机的行驶里程等，以便更好地了解运输情况和优化运输效率。
- 金融企业的交易分析：统计每个客户的交易总额、每个产品的销售量、每个交易员的业绩等，以便更好地了解交易情况和优化业务流程。
- 智能家居的设备监控分析：统计每个设备的使用次数、每个家庭的能源消耗量、每个时间段的设备使用率等，以便更好地了解用户需求和优化设备效能。

基本语法

聚合查询的语法结构与其他查询相似，通常包含以下部分：

- 查询条件：指定需要聚合的文档，可以使用标准的 Elasticsearch 查询语法，如 term、match、range 等等。
- 聚合函数：指定要执行的聚合操作，如 sum、avg、min、max、terms、date_histogram 等等。每个聚合命令都会生成一个聚合结果。
- 聚合嵌套：聚合命令可以嵌套，以便更细粒度地分析数据。

```
GET <index_name>/_search
{
  "aggs": {
    "<aggs_name>": { // 聚合名称需要自己定义
      "<agg_type>": {
        "field": "<field_name>"
      }
    }
  }
}
```

- aggs_name：聚合函数的名称
- agg_type：聚合种类，比如是桶聚合（terms）或者是指标聚合（avg、sum、min、max等）
- field_name：字段名称或者叫域名。

4.1 聚合的分类

- Metric Aggregation：一些数学运算，可以对文档字段进行统计分析，类比Mysql中的 min(), max(), sum() 操作。

```
SELECT MIN(price), MAX(price) FROM products
#Metric聚合的DSL类比实现：
{
    "aggs":{
        "avg_price":{
            "avg":{
                "field":"price"
            }
        }
    }
}
```

- Bucket Aggregation： 一些满足特定条件的文档的集合放置到一个桶里，每一个桶关联一个key，类比Mysql中的group by操作。

```
ELECT size COUNT(*) FROM products GROUP BY size
#bucket聚合的DSL类比实现：
{
 "aggs": {
    "by_size": {
      "terms": {
        "field": "size"
      }
  }
}
```

- Pipeline Aggregation：对其他的聚合结果进行二次聚合

示例数据

```
DELETE /employees
#创建索引库
PUT /employees
```

```
{
  "mappings": {
    "properties": {
      "age":{
        "type": "integer"
      },
      "gender":{
        "type": "keyword"
      },
      "job":{
         "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 50
            }
          }
      },
      "name":{
        "type": "keyword"
      },
      "salary":{
        "type": "integer"
      }
    }
  }
}

PUT /employees/_bulk
{ "index" : {  "_id" : "1" } }
{ "name" : "Emma","age":32,"job":"Product Manager","gender":"female","salary":35000 }
{ "index" : {  "_id" : "2" } }
{ "name" : "Underwood","age":41,"job":"Dev Manager","gender":"male","salary": 50000}
{ "index" : {  "_id" : "3" } }
{ "name" : "Tran","age":25,"job":"Web Designer","gender":"male","salary":18000 }
{ "index" : {  "_id" : "4" } }
{ "name" : "Rivera","age":26,"job":"Web Designer","gender":"female","salary": 22000}
{ "index" : {  "_id" : "5" } }
{ "name" : "Rose","age":25,"job":"QA","gender":"female","salary":18000 }
{ "index" : {  "_id" : "6" } }
```

```
{ "name" : "Lucy","age":31,"job":"QA","gender":"female","salary": 25000}
{ "index" : {  "_id" : "7" } }
{ "name" : "Byrd","age":27,"job":"QA","gender":"male","salary":20000 }
{ "index" : {  "_id" : "8" } }
{ "name" : "Foster","age":27,"job":"Java Programmer","gender":"male","salary": 20000}
{ "index" : {  "_id" : "9" } }
{ "name" : "Gregory","age":32,"job":"Java Programmer","gender":"male","salary":22000 }
{ "index" : {  "_id" : "10" } }
{ "name" : "Bryant","age":20,"job":"Java Programmer","gender":"male","salary": 9000}
{ "index" : {  "_id" : "11" } }
{ "name" : "Jenny","age":36,"job":"Java Programmer","gender":"female","salary":38000 }
{ "index" : {  "_id" : "12" } }
{ "name" : "Mcdonald","age":31,"job":"Java Programmer","gender":"male","salary": 32000}
{ "index" : {  "_id" : "13" } }
{ "name" : "Jonthna","age":30,"job":"Java Programmer","gender":"female","salary":30000
}
{ "index" : {  "_id" : "14" } }
{ "name" : "Marshall","age":32,"job":"Javascript Programmer","gender":"male","salary":
25000}
{ "index" : {  "_id" : "15" } }
{ "name" : "King","age":33,"job":"Java Programmer","gender":"male","salary":28000 }
{ "index" : {  "_id" : "16" } }
{ "name" : "Mccarthy","age":21,"job":"Javascript Programmer","gender":"male","salary":
16000}
{ "index" : {  "_id" : "17" } }
{ "name" : "Goodwin","age":25,"job":"Javascript Programmer","gender":"male","salary":
16000}
{ "index" : {  "_id" : "18" } }
{ "name" : "Catherine","age":29,"job":"Javascript
Programmer","gender":"female","salary": 20000}
{ "index" : {  "_id" : "19" } }
{ "name" : "Boone","age":30,"job":"DBA","gender":"male","salary": 30000}
{ "index" : {  "_id" : "20" } }
{ "name" : "Kathy","age":29,"job":"DBA","gender":"female","salary": 20000}
```

4.2 Metric Aggregation

- 单值分析：只输出一个分析结果
  - min, max, avg, sum

    - Cardinality（类似distinct Count)
- 多值分析:输出多个分析结果
    - stats（统计）, extended stats
    - percentile （百分位）, percentile rank
    - top hits(排在前面的示例)

查询员工的最低最高和平均工资

```
#多个 Metric 聚合，找到最低最高和平均工资
POST /employees/_search
{
  "size": 0,
  "aggs": {
    "max_salary": {
      "max": {
        "field": "salary"
      }
    },
    "min_salary": {
      "min": {
        "field": "salary"
      }
    },
    "avg_salary": {
      "avg": {
        "field": "salary"
      }
    }
  }
}
```

对salary进行统计

```
# 一个聚合，输出多值
POST /employees/_search
{
```

```
  "size": 0,
  "aggs": {
    "stats_salary": {
      "stats": {
        "field":"salary"
      }
    }
  }
}
```

cardinate对搜索结果去重

```
POST /employees/_search
{
  "size": 0,
  "aggs": {
    "cardinate": {
      "cardinality": {
        "field": "job.keyword"
      }
    }
  }
}
```

4.3 Bucket Aggregation

按照一定的规则，将文档分配到不同的桶中，从而达到分类的目的。ES提供的一些常见的 Bucket Aggregation。

- Terms，需要字段支持filedata
  - keyword 默认支持fielddata
  - text需要在Mapping 中开启fielddata，会按照分词后的结果进行分桶
- 数字类型

- Range / Data Range
- Histogram（直方图） / Date Histogram

- 支持嵌套: 也就在桶里再做分桶

获取job的分类信息

```
# 对keword 进行聚合
GET /employees/_search
{
  "size": 0,
  "aggs": {
    "jobs": {
      "terms": {
        "field":"job.keyword"
      }
    }
  }
}
```

聚合可配置属性有：

- field：指定聚合字段
- size：指定聚合结果数量
- order：指定聚合结果排序方式

默认情况下，Bucket聚合会统计Bucket内的文档数量，记为_count，并且按照_count降序排序。我们可以指定order属性，自定义聚合的排序方式：

```
GET /employees/_search
{
  "size": 0,
  "aggs": {
    "jobs": {
      "terms": {
        "field":"job.keyword",
         "size": 10,
        "order": {
          "_count": "desc"
```

```
        }
      }
    }
  }
}
```

限定聚合范围

```
#只对salary在10000元以上的文档聚合
GET /employees/_search
{
  "query": {
    "range": {
      "salary": {
        "gte": 10000
      }
    }
  },
  "size": 0,
  "aggs": {
    "jobs": {
      "terms": {
        "field":"job.keyword",
         "size": 10,
        "order": {
          "_count": "desc"
        }
      }
    }
  }
}
```

注意：对 Text 字段进行 terms 聚合查询，会失败抛出异常

```
POST /employees/_search
```

```
{
  "size": 0,
  "aggs": {
    "jobs": {
      "terms": {
        "field":"job"
      }
    }
  }
}
```

解决办法：对 Text 字段打开 fielddata，支持terms aggregation

```
PUT /employees/_mapping
{
  "properties" : {
    "job":{
       "type":  "text",
       "fielddata": true
    }
  }
}

# 对 Text 字段进行分词，分词后的terms
POST /employees/_search
{
  "size": 0,
  "aggs": {
    "jobs": {
      "terms": {
        "field":"job"
      }
    }
  }
}
```

对job.keyword 和 job 进行 terms 聚合，分桶的总数并不一样

```
POST /employees/_search
{
  "size": 0,
  "aggs": {
    "cardinate": {
      "cardinality": {
        "field": "job"
      }
    }
  }
}
```

Range & Histogram聚合

- 按照数字的范围，进行分桶
- 在Range Aggregation中，可以自定义Key

Range 示例：按照工资的 Range 分桶

```
Salary Range分桶，可以自己定义 key
POST employees/_search
{
  "size": 0,
  "aggs": {
    "salary_range": {
      "range": {
        "field":"salary",
        "ranges":[
          {
            "to":10000
          },
          {
            "from":10000,
```

```
            "to":20000
          },
          {
            "key":">20000",
            "from":20000
          }
        ]
      }
    }
  }
}
```

Histogram示例：按照工资的间隔分桶

```
#工资0到10万，以 5000一个区间进行分桶
POST employees/_search
{
  "size": 0,
  "aggs": {
    "salary_histrogram": {
      "histogram": {
        "field":"salary",
        "interval":5000,
        "extended_bounds":{
          "min":0,
          "max":100000
        }
      }
    }
  }
}
```

top_hits应用场景: 当获取分桶后，桶内最匹配的顶部文档列表

```
# 指定size，不同工种中，年纪最大的3个员工的具体信息
POST /employees/_search
{
  "size": 0,
  "aggs": {
    "jobs": {
      "terms": {
        "field":"job.keyword"
      },
      "aggs":{
        "old_employee":{
          "top_hits":{
            "size":3,
            "sort":[
              {
                "age":{
                  "order":"desc"
                }
              }
            ]
          }
        }
      }
    }
  }
}

```

嵌套聚合示例

```
# 嵌套聚合1，按照工作类型分桶，并统计工资信息
POST employees/_search
{
  "size": 0,
  "aggs": {
```

```
    "Job_salary_stats": {
      "terms": {
        "field": "job.keyword"
      },
      "aggs": {
        "salary": {
          "stats": {
            "field": "salary"
          }
        }
      }
    }
  }
}

# 多次嵌套。根据工作类型分桶，然后按照性别分桶，计算工资的统计信息
POST employees/_search
{
  "size": 0,
  "aggs": {
    "Job_gender_stats": {
      "terms": {
        "field": "job.keyword"
      },
      "aggs": {
        "gender_stats": {
          "terms": {
            "field": "gender"
          },
          "aggs": {
            "salary_stats": {
              "stats": {
                "field": "salary"
              }
            }
          }
        }
      }
    }
  }
```

```
}
```

4.4 Pipeline Aggregation

支持对聚合分析的结果，再次进行聚合分析。

Pipeline 的分析结果会输出到原结果中，根据位置的不同，分为两类：

- Sibling - 结果和现有分析结果同级
  - Max，min，Avg & Sum Bucket
  - Stats，Extended Status Bucket
  - Percentiles Bucket
- Parent -结果内嵌到现有的聚合分析结果之中
  - Derivative(求导)
  - Cumultive Sum(累计求和)
  - Moving Function(移动平均值)

min_bucket示例

在员工数最多的工种里，找出平均工资最低的工种

```
# 平均工资最低的工种
POST employees/_search
{
  "size": 0,
  "aggs": {
    "jobs": {
      "terms": {
        "field": "job.keyword",
        "size": 10
      },
      "aggs": {
        "avg_salary": {
          "avg": {
            "field": "salary"
          }
        }
      }
```

```
    },
    "min_salary_by_job":{
      "min_bucket": {
        "buckets_path": "jobs>avg_salary"
      }
    }
  }
}
```

- min_salary_by_job结果和jobs的聚合同级
- min_bucket求之前结果的最小值
- 通过bucket_path关键字指定路径

Stats示例

```
# 平均工资的统计分析
POST employees/_search
{
  "size": 0,
  "aggs": {
    "jobs": {
      "terms": {
        "field": "job.keyword",
        "size": 10
      },
      "aggs": {
        "avg_salary": {
          "avg": {
            "field": "salary"
          }
        }
      }
    },
    "stats_salary_by_job":{
      "stats_bucket": {
        "buckets_path": "jobs>avg_salary"
      }
    }
  }
```

```
}
```

percentiles示例

```
# 平均工资的百分位数
POST employees/_search
{
  "size": 0,
  "aggs": {
    "jobs": {
      "terms": {
        "field": "job.keyword",
        "size": 10
      },
      "aggs": {
        "avg_salary": {
          "avg": {
            "field": "salary"
          }
        }
      }
    },
    "percentiles_salary_by_job":{
      "percentiles_bucket": {
        "buckets_path": "jobs>avg_salary"
      }
    }
  }
}

```

Cumulative_sum示例

```
#Cumulative_sum    累计求和
POST employees/_search
```

```
{
  "size": 0,
  "aggs": {
    "age": {
      "histogram": {
        "field": "age",
        "min_doc_count": 0,
        "interval": 1
      },
      "aggs": {
        "avg_salary": {
          "avg": {
            "field": "salary"
          }
        },
        "cumulative_salary":{
          "cumulative_sum": {
            "buckets_path": "avg_salary"
          }
        }
      }
    }
  }
}

```

4.5 聚合的作用范围

ES聚合分析的默认作用范围是query的查询结果集，同时ES还支持以下方式改变聚合的作用范围：

- Filter
- Post Filter
- Global

```
#Query
POST employees/_search
{
```

```
  "size": 0,
  "query": {
    "range": {
      "age": {
        "gte": 20
      }
    }
  },
  "aggs": {
    "jobs": {
      "terms": {
        "field":"job.keyword"

      }
    }
  }
}

#Filter
POST employees/_search
{
  "size": 0,
  "aggs": {
    "older_person": {
      "filter":{
        "range":{
          "age":{
            "from":35
          }
        }
      },
      "aggs":{
         "jobs":{
           "terms": {
        "field":"job.keyword"
      }
      }
    }},
    "all_jobs": {
      "terms": {
```

```
        "field":"job.keyword"

      }
    }
  }
}


#Post field. 一条语句，找出所有的job类型。还能找到聚合后符合条件的结果
POST employees/_search
{
  "aggs": {
    "jobs": {
      "terms": {
        "field": "job.keyword"
      }
    }
  },
  "post_filter": {
    "match": {
      "job.keyword": "Dev Manager"
    }
  }
}


#global
POST employees/_search
{
  "size": 0,
  "query": {
    "range": {
      "age": {
        "gte": 40
      }
    }
  },
  "aggs": {
```

```
    "jobs": {
      "terms": {
        "field":"job.keyword"

      }
    },

    "all":{
      "global":{},
      "aggs":{
        "salary_avg":{
          "avg":{
            "field":"salary"
          }
        }
      }
    }
  }
}


```

4.6 排序

指定order，按照count和key进行排序：

- 默认情况，按照count降序排序
- 指定size，就能返回相应的桶

```
#排序 order
#count and key
POST employees/_search
{
  "size": 0,
  "query": {
    "range": {
      "age": {
```

```
        "gte": 20
      }
    }
  },
  "aggs": {
    "jobs": {
      "terms": {
        "field":"job.keyword",
        "order":[
          {"_count":"asc"},
          {"_key":"desc"}
          ]

      }
    }
  }
}


#排序 order
#count and key
POST employees/_search
{
  "size": 0,
  "aggs": {
    "jobs": {
      "terms": {
        "field":"job.keyword",
        "order":[  {
            "avg_salary":"desc"
          }]


      },
    "aggs": {
      "avg_salary": {
        "avg": {
          "field":"salary"
        }
```

```
      }
    }
    }
  }
}


#排序 order
#count and key
POST employees/_search
{
  "size": 0,
  "aggs": {
    "jobs": {
      "terms": {
        "field":"job.keyword",
        "order":[  {
            "stats_salary.min":"desc"
          }]


      },
    "aggs": {
      "stats_salary": {
        "stats": {
          "field":"salary"
        }
      }
    }
    }
  }
}
```

4.7 ES聚合分析不精准原因分析

ElasticSearch在对海量数据进行聚合分析的时候会损失搜索的精准度来满足实时性的需求。

Terms聚合分析的执行流程：

不精准的原因： 数据分散到多个分片，聚合是每个分片的取 Top X，导致结果不精准。ES 可以不每个分片Top X，而是全量聚合，但势必这会有很大的性能问题。

思考：如何提高聚合精确度?

方案1：设置主分片为1

注意7.x版本已经默认为1。

适用场景：数据量小的小集群规模业务场景。

方案2：调大 shard_size 值

设置 shard_size 为比较大的值，官方推荐：size*1.5+10。shard_size 值越大，结果越趋近于精准聚合结果值。此外，还可以通过show_term_doc_count_error参数显示最差情况下的错误值，用于辅助确定 shard_size 大小。

- size：是聚合结果的返回值，客户期望返回聚合排名前三，size值就是 3。
- shard_size: 每个分片上聚合的数据条数。shard_size 原则上要大于等于 size

适用场景：数据量大、分片数多的集群业务场景。

测试： 使用kibana的测试数据

```
DELETE my_flights
PUT my_flights
{
  "settings": {
    "number_of_shards": 20
  },
  "mappings" : {
      "properties" : {
        "AvgTicketPrice" : {
          "type" : "float"
        },
        "Cancelled" : {
          "type" : "boolean"
        },
        "Carrier" : {
          "type" : "keyword"
        },
        "Dest" : {
```

```
          "type" : "keyword"
        },
        "DestAirportID" : {
          "type" : "keyword"
        },
        "DestCityName" : {
          "type" : "keyword"
        },
        "DestCountry" : {
          "type" : "keyword"
        },
        "DestLocation" : {
          "type" : "geo_point"
        },
        "DestRegion" : {
          "type" : "keyword"
        },
        "DestWeather" : {
          "type" : "keyword"
        },
        "DistanceKilometers" : {
          "type" : "float"
        },
        "DistanceMiles" : {
          "type" : "float"
        },
        "FlightDelay" : {
          "type" : "boolean"
        },
        "FlightDelayMin" : {
          "type" : "integer"
        },
        "FlightDelayType" : {
          "type" : "keyword"
        },
        "FlightNum" : {
          "type" : "keyword"
        },
        "FlightTimeHour" : {
          "type" : "keyword"
```

```
        },
        "FlightTimeMin" : {
          "type" : "float"
        },
        "Origin" : {
          "type" : "keyword"
        },
        "OriginAirportID" : {
          "type" : "keyword"
        },
        "OriginCityName" : {
          "type" : "keyword"
        },
        "OriginCountry" : {
          "type" : "keyword"
        },
        "OriginLocation" : {
          "type" : "geo_point"
        },
        "OriginRegion" : {
          "type" : "keyword"
        },
        "OriginWeather" : {
          "type" : "keyword"
        },
        "dayOfWeek" : {
          "type" : "integer"
        },
        "timestamp" : {
          "type" : "date"
        }
      }
    }
}

POST _reindex
{
  "source": {
    "index": "kibana_sample_data_flights"
```

```
  },
  "dest": {
    "index": "my_flights"
  }
}

GET my_flights/_count
GET kibana_sample_data_flights/_search
{
  "size": 0,
  "aggs": {
    "weather": {
      "terms": {
        "field":"OriginWeather",
        "size":5,
        "show_term_doc_count_error":true
      }
    }
  }
}

GET my_flights/_search
{
  "size": 0,
  "aggs": {
    "weather": {
      "terms": {
        "field":"OriginWeather",
        "size":5,
        "shard_size":10,
        "show_term_doc_count_error":true
      }
    }
  }
}
```

在Terms Aggregation的返回中有两个特殊的数值：

- doc_count_error_upper_bound : 被遗漏的term 分桶，包含的文档，有可能的最大值
- sum_other_doc_count: 除了返回结果 bucket的terms以外，其他 terms 的文档总数（总数-返回的总数)

方案3：将size设置为全量值，来解决精度问题

将size设置为2的32次方减去1也就是分片支持的最大值，来解决精度问题。

原因：1.x版本，size等于 0 代表全部，高版本取消 0 值，所以设置了最大值（大于业务的全量值）。

全量带来的弊端就是：如果分片数据量极大，这样做会耗费巨大的CPU 资源来排序，而且可能会阻塞网络。

适用场景：对聚合精准度要求极高的业务场景，由于性能问题，不推荐使用。

方案4：使用Clickhouse/ Spark 进行精准聚合

适用场景：数据量非常大、聚合精度要求高、响应速度快的业务场景。

4.8 Elasticsearch 聚合性能优化

启用 eager global ordinals 提升高基数聚合性能

适用场景：高基数聚合 。高基数聚合场景中的高基数含义：一个字段包含很大比例的唯一值。

global ordinals 中文翻译成全局序号，是一种数据结构，应用场景如下：

- 基于 keyword，ip 等字段的分桶聚合，包含：terms聚合、composite 聚合等。
- 基于text 字段的分桶聚合（前提条件是：fielddata 开启）。
- 基于父子文档 Join 类型的 has_child 查询和 父聚合。

global ordinals 使用一个数值代表字段中的字符串值，然后为每一个数值分配一个 bucket（分桶）。

global ordinals 的本质是：启用 eager_global_ordinals 时，会在刷新（refresh）分片时构建全局序号。这将构建全局序号的成本从搜索阶段转移到了数据索引化（写入）阶段。

创建索引的同时开启：eager_global_ordinals。

```
PUT /my-index
{
  "mappings": {
    "properties": {
      "tags": {
        "type": "keyword",
        "eager_global_ordinals": true
      }
    }
```

```
  }
```

注意：开启 eager_global_ordinals 会影响写入性能，因为每次刷新时都会创建新的全局序号。为了最大程度地减少由于频繁刷新建立全局序号而导致的额外开销，请调大刷新间隔 refresh_interval。

动态调整刷新频率的方法如下：

```
PUT my-index/_settings
{
  "index": {
    "refresh_interval": "30s"
  }
```

该招数的本质是：以空间换时间。

插入数据时对索引进行预排序

- Index sorting （索引排序）可用于在插入时对索引进行预排序，而不是在查询时再对索引进行排序，这将提高范围查询（range query）和排序操作的性能。
- 在 Elasticsearch 中创建新索引时，可以配置如何对每个分片内的段进行排序。
- 这是 Elasticsearch 6.X 之后版本才有的特性。

```

PUT /my_index
{
  "settings": {
    "index":{
      "sort.field": "create_time",
      "sort.order": "desc"
    }
  },
  "mappings": {
    "properties": {
      "create_time":{
        "type": "date"
      }
    }
```

```
  }
}
```

注意：预排序将增加 Elasticsearch 写入的成本。在某些用户特定场景下，开启索引预排序会导致大约 40%-50% 的写性能下降。也就是说，如果用户场景更关注写性能的业务，开启索引预排序不是一个很好的选择。

使用节点查询缓存

节点查询缓存（Node query cache）可用于有效缓存过滤器（filter）操作的结果。如果多次执行同一 filter 操作，这将很有效，但是即便更改过滤器中的某一个值，也将意味着需要计算新的过滤器结果。

例如，由于 “now” 值一直在变化，因此无法缓存在过滤器上下文中使用 “now” 的查询。

那怎么使用缓存呢？通过在 now 字段上应用 datemath 格式将其四舍五入到最接近的分钟/小时等，可以使此类请求更具可缓存性，以便可以对筛选结果进行缓存。

```
PUT /my_index/_doc/1
{
  "create_time":"2022-05-11T16:30:55.328Z"
}

#下面的示例无法使用缓存
GET /my_index/_search
{
  "query":{
    "constant_score": {
      "filter": {
        "range": {
          "create_time": {
            "gte": "now-1h",
            "lte": "now"
          }
        }
      }
    }
  }
}

```

```
# 下面的示例就可以使用节点查询缓存。
GET /my_index/_search
{
  "query":{
    "constant_score": {
      "filter": {
        "range": {
          "create_time": {
            "gte": "now-1h/m",
            "lte": "now/m"
          }
        }
      }
    }
  }
}
```

上述示例中的“now-1h/m” 就是 datemath 的格式。
如果当前时间 now 是：16:31:29，那么range query 将匹配 my_date 介于：15:31:00 和 15:31:59 之间的时间数据。同理，聚合的前半部分 query 中如果有基于时间查询，或者后半部分 aggs 部分中有基于时间聚合的，建议都使用 datemath 方式做缓存处理以优化性能。

使用分片请求缓存

聚合语句中，设置：size：0，就会使用分片请求缓存缓存结果。size = 0 的含义是：只返回聚合结果，不返回查询结果。

```
GET /es_db/_search
{
  "size": 0,
  "aggs": {
    "remark_agg": {
      "terms": {
        "field": "remark.keyword"
      }
    }
  }
}
```

拆分聚合，使聚合并行化

Elasticsearch 查询条件中同时有多个条件聚合，默认情况下聚合不是并行运行的。当为每个聚合提供自己的查询并执行 msearch 时，性能会有显著提升。因此，在 CPU 资源不是瓶颈的前提下，如果想缩短响应时间，可以将多个聚合拆分为多个查询，借助：msearch 实现并行聚合。

```
#常规的多条件聚合实现
GET /employees/_search
{
  "size": 0,
  "aggs": {
    "job_agg": {
      "terms": {
        "field": "job.keyword"
      }
    },
    "max_salary":{
      "max": {
        "field": "salary"
      }
    }
  }
}
# msearch 拆分多个语句的聚合实现
GET _msearch
{"index":"employees"}
{"size":0,"aggs":{"job_agg":{"terms":{"field": "job.keyword"}}}}
{"index":"employees"}
{"size":0,"aggs":{"max_salary":{"max":{"field": "salary"}}}}
```