主讲老师：Fox

```
文档：3. ElasticSearch搜索技术深入与聚合查...
链接：http://note.youdao.com/noteshare?id=44848fef10c56f33bdb7a85f7ca04376&sub=89B65B40D9584654978759587D3ADEB9
```

相关性和相关性算分

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

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

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。ES 5之前，默认的相关性算分采用TF-IDF，现在采用BM 25。

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

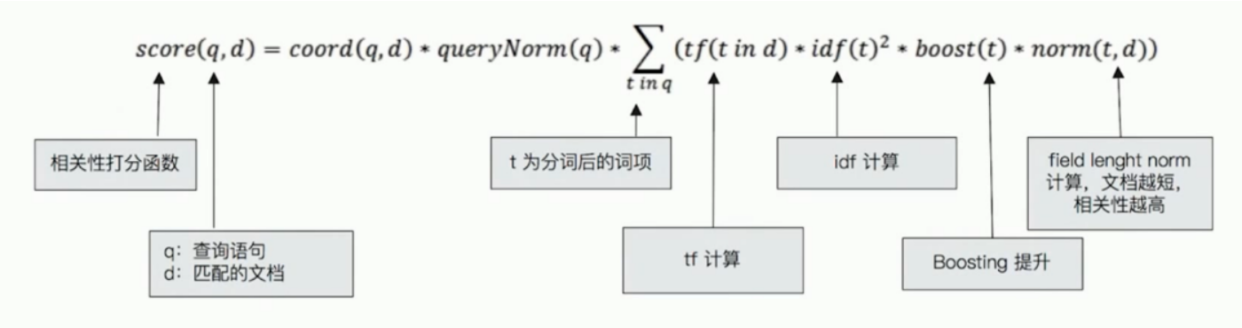
JAVA	1,2,3
设计模式	1,2,3,4,5,6
多线程	2,3,7,9

什么是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。

- **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。

- **字段长度归一值（ field-length norm)**

字段的长度是多少？字段越短，字段的权重越高。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）、逆向文档频率（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

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

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"
 }
 }
}
```

Boosting

Boosting是控制相关度的一种手段。

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

返回匹配positive查询的文档并降低匹配negative查询的文档相似度分。这样就可以在不排除某些文档的前提下对文档进行查询,搜索结果中存在只不过相似度分数相比正常匹配的要低;

```
GET /test_score/_search
{
 "query": {
 "boosting": {
 "positive": {
 "term": {
 "content": "elasticsearch"
 }
 },
 "negative": {
 "term": {
 "content": "like"
 }
 },
 "negative_boost": 0.2
```

```
  }
  }
}
```

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。

布尔查询bool Query

一个bool查询,是一个或者多个查询子句的组合，总共包括4种子句，其中2种会影响算分，2种不影响算分。

- must: 相当于&& ，必须匹配，贡献算分
- should: 相当于|| ，选择性匹配，贡献算分
- must_not: 相当于! ，必须不能匹配，不贡献算分
- filter: 必须匹配，不贡献算法

在Elasticsearch中，有Query和 Filter两种不同的Context

- Query Context: 相关性算分
- Filter Context: 不需要算分 ,可以利用Cache，获得更好的性能

相关性并不只是全文本检索的专利，也适用于yes | no 的子句，匹配的子句越多，相关性评分
越高。如果多条查询子句被合并为一条复合查询语句，比如 bool查询，则每个查询子句计算得出的评分会被合并到总的相关性评分中。

bool查询语法

- 子查询可以任意顺序出现
- 可以嵌套多个查询
- 如果你的bool查询中，没有must条件,should中必须至少满足一条查询

```
GET /es_db/_search
{
 "query": {
 "bool": {
 "must": {
 "match": {
 "remark": "java developer"
 }
 },
  "filter": {
  "term": {
```

```
  "sex": "1"
  }
  },
  "must_not": {
  "range": {
  "age": {
  "gte": 30
  }
  }
  },
  "should": [
  {
  "term": {
  "address.keyword": {
  "value": "广州天河公园"
  }
  }
  },
  {
  "term": {
  "address.keyword": {
  "value": "广州白云山公园"
  }
  }
  }
  ],
  "minimum_should_match": 1
  }
  }
}
```

如何解决结构化查询“包含而不是相等”的问题

测试数据

```
POST /employee/_bulk
{"index":{"_id":1}}
{"name":"小明","interest":["跑步","篮球"]}
{"index":{"_id":2}}
{"name":"小红","interest":["跑步"]}
{"index":{"_id":3}}
{"name":"小丽","interest":["跳舞","唱歌","跑步"]}
```

```
8
9  POST /employee/_search
10 {
11  "query": {
12  "term": {
13  "interest.keyword": {
14  "value": "跑步"
15  }
16  }
17  }
18 }
```

解决方案： 增加count字段，使用bool查询解决

- 从业务角度，按需改进Elasticsearch数据模型

```
1  POST /employee/_bulk
2  {"index":{"_id":1}}
3  {"name":"小明","interest":["跑步","篮球"],"interest_count":2}
4  {"index":{"_id":2}}
5  {"name":"小红","interest":["跑步"],"interest_count":1}
6  {"index":{"_id":3}}
7  {"name":"小丽","interest":["跳舞","唱歌","跑步"],"interest_count":3}
```

- 使用bool查询

```
1  # must 算分
2  POST /employee/_search
3  {
4   "query": {
5   "bool": {
6   "must": [
7   {
8   "term": {
9   "interest.keyword": {
10   "value": "跑步"
11   }
12   }
13   },
14   {
15   "term": {
16   "interest_count": {
17   "value": 1
18   }
```

```
  }
  }
  ]
  }
  }
}
# filter不算分
POST /employee/_search
{
 "query": {
 "bool": {
 "filter": [
 {
 "term": {
 "interest.keyword": {
 "value": "跑步"
 }
 }
 },
 {
 "term": {
 "interest_count": {
 "value": 1
 }
 }
 }
 ]
 }
 }
}
```

利用bool嵌套实现should not逻辑

```
GET /es_db/_search
{
 "query": {
 "bool": {
 "must": {
 "match": {
 "remark": "java developer"
 }
```

```
  },
   "should": [
   {
   "bool": {
   "must_not": [
   {
   "term": {
   "sex": 1
   }
   }
   ]
   }
   }
   ],
   "minimum_should_match": 1
   }
   }
}
```

Boosting Query

思考： 如何控制查询的相关性算分?

控制字段的Boosting

Boosting是控制相关的一种手段。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

```
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

- 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

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。

```
GET /news/_search
{
 "query": {
```

```
"boosting": {
"positive": {
"match": {
"content": "apple"
}
},
 "negative": {
 "match": {
 "content": "pie"
 }
 },
 "negative_boost": 0.2
 }
 }
}
```

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

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

```
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？

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，不应该讲分数简单叠加，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": 0.2
  }
  }
}
```

Multi Match Query

最佳字段(Best Fields)搜索

Best Fields是默认类型，可以不用指定

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

使用多数字段（Most Fields）搜索

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"
  ]
  }
  }
}
```

跨字段（Cross Field）搜索

```
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

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"
 }
 }
 }
}

GET /address/_search
{
 "query": {
 "multi_match": {
 "query": "湖南常德",
 "type": "most_fields",
 "fields": ["province","city"]
 }
 }
}
```

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?

语法:

```
"aggs" : { #和query同级的关键词
 "<aggregation_name>" : { #自定义的聚合名字
 "<aggregation_type>" : { #聚合的定义:  不同的type+body
 <aggregation_body>
 }
 [,"meta" : { [<meta_data_body>] } ]?
 [,"aggregations" : { [<sub_aggregation>]+ } ]? #子聚合查询
 }
 [,"<aggregation_name_2>" : { ... } ]* #可以包含多个同级的聚合查询
}
```

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}
```

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"
 }
 }
  }
}
```

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"
 }
 }
```

```
  }
}
```

```
{
  "error" : {
    "root_cause" : [
      {
        "type" : "illegal_argument_exception",
        "reason" : "Text fields are not optimised for operations that require per-document
          field data like aggregations and sorting, so these operations are disabled by
          default. Please use a keyword field instead. Alternatively, set fielddata=true on
          [job] in order to load field data by uninverting the inverted index. Note that
          this can use significant memory."
      }
    ],
    "type" : "search_phase_execution_exception",
    "reason" : "all shards failed",
    "phase" : "query",
    "grouped" : true,
    "failed_shards" : [
```

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"
  }
  }
  }
  }
  }
  }
}
```

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"
  }
  }
  }
  }
  }
 }

```

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"
  }
  }
  }
  }
   }
}


```

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

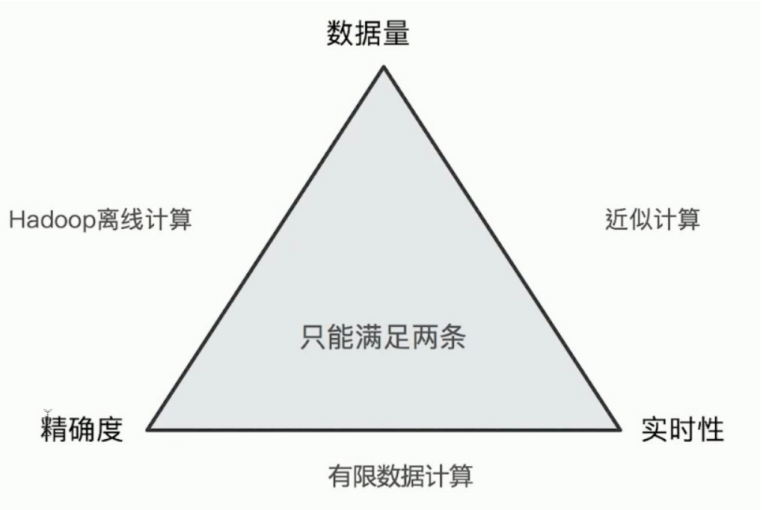
```
#排序 order
#count and key
POST employees/_search
{
 "size": 0,
 "query": {
 "range": {
 "age": {
 "gte": 20
  }
  }
  },
```

```
  "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"
  }
  }
  }
  }
  }
}

```

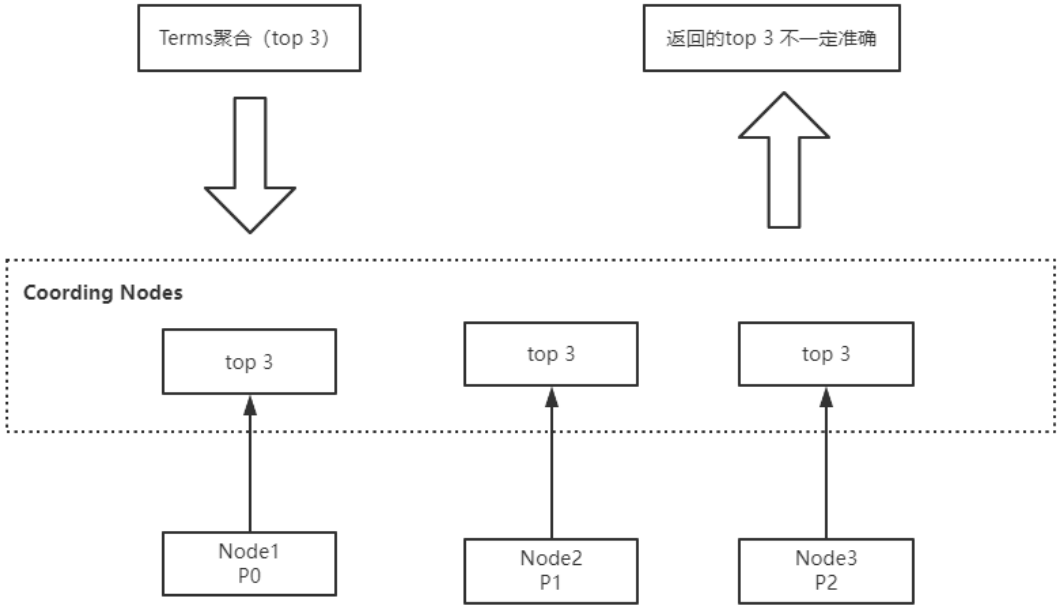
```
54
55 #排序 order
56 #count and key
57 POST employees/_search
58 {
59  "size": 0,
60  "aggs": {
61  "jobs": {
62  "terms": {
63  "field":"job.keyword",
64  "order":[ {
65  "stats_salary.min":"desc"
66  }]
67
68
69  },
70  "aggs": {
71  "stats_salary": {
72  "stats": {
73  "field":"salary"
74  }
75  }
76  }
77  }
78  }
79 }
```

ES聚合分析不精准原因分析

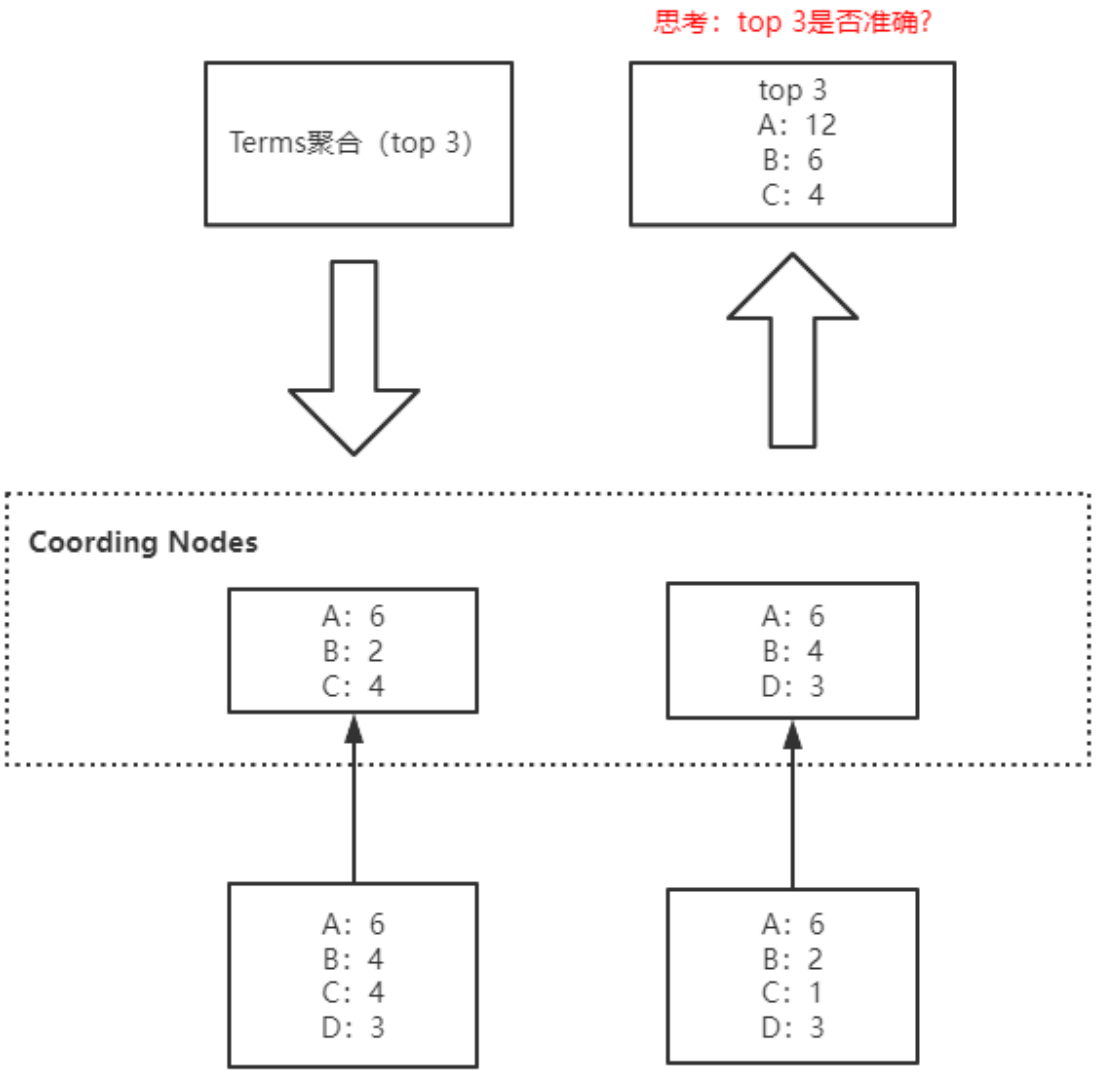
ElasticSearch在对海量数据进行聚合分析的时候会损失搜索的精准度来满足实时性的需求。



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。

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"}
```

```
23 {"size":0,"aggs":{"max_salary":{"max":{"field": "salary"}}}}
```