### 查询

1、query string search 2、query DSL 3、query filter 4、full-text search 5、phrase search 6、highlight search

### 1, query string search

搜索全部商品: GET /ecommerce/product/\_search

● took: 耗费了几毫秒

• timed\_out: 是否超时, 这里是没有

- \_shards:数据拆成了5个分片,所以对于搜索请求,会打到所有的primary shard(或者是它的某个replica shard也可以)
- hits.total: 查询结果的数量, 3个document
- hits.max\_score: score的含义,就是document对于一个search的相关度的匹配分数,越相关,就越匹配,分数也高
- hits.hits: 包含了匹配搜索的document的详细数据

```
1
   {
 2
      "took": 2,
 3
      "timed out": false,
      "_shards": {
 4
        "total": 5,
 5
        "successful": 5,
 6
        "failed": 0
 7
 8
      },
      "hits": {
 9
        "total": 3,
10
         "max_score": 1,
11
        "hits": [
12
13
          {
             "_index": "ecommerce",
14
            "_type": "product",
15
            " id": "2",
16
17
             " score": 1,
             "_source": {
18
               "name": "jiajieshi yagao",
19
20
               "desc": "youxiao fangzhu",
               "price": 25,
21
               "producer": "jiajieshi producer",
22
               "tags": [
23
                 "fangzhu"
24
25
               ]
26
```

```
27
           },
28
           {
29
             " index": "ecommerce",
             "_type": "product",
30
31
             " id": "1",
             " score": 1,
32
3.3
             " source": {
               "name": "gaolujie yagao",
34
35
               "desc": "gaoxiao meibai",
               "price": 30,
36
               "producer": "gaolujie producer",
37
               "tags": [
38
                 "meibai",
39
                 "fangzhu"
40
41
               1
42
             }
           },
43
44
             "_index": "ecommerce",
45
             "_type": "product",
46
47
             " id": "3",
             " score": 1,
48
49
             " source": {
               "name": "zhonghua yagao",
50
51
               "desc": "caoben zhiwu",
               "price": 40,
52
               "producer": "zhonghua producer",
53
               "tags": [
54
                 "qingxin"
55
56
               ]
57
             }
58
           }
59
60
      }
61
    }
```

query string search的由来,因为search参数都是以http请求的query string来附带的

搜索商品名称中包含yagao的商品,而且按照售价降序排序:GET /ecommerce/product/\_search?q=name:yagao&sort=price:desc

适用于临时的在命令行使用一些工具,比如curl,快速的发出请求,来检索想要的信息;但是如果查询请求很复杂,是很难去构建的 在生产环境中,几乎很少使用query string search

### 2、query DSL

DSL: Domain Specified Language,特定领域的语言 http request body: 请求体,可以用 json的格式来构建查询语法,比较方便,可以构建各种复杂的语法,比query string search肯定强大多了

```
1 GET /ecommerce/product/_search
2 {
3    "query": { "match_all": {} }
4  }
```

查询名称包含yagao的商品,同时按照价格降序排序

```
GET /ecommerce/product/ search
2
      "query" : {
3
4
           "match" : {
             "name" : "yagao"
5
          }
6
7
      },
      "sort": [
          { "price": "desc" }
9
10
      ]
11 }
```

分页查询商品,总共3条商品,假设每页就显示1条商品,现在显示第2页,所以就查出来第2个商品

```
1  GET /ecommerce/product/_search
2  {
3    "query": { "match_all": {} },
4    "from": 1,
5    "size": 1
6  }
```

指定要查询出来商品的名称和价格就可以

```
1  GET /ecommerce/product/_search
2  {
3    "query": { "match_all": {} },
4    "_source": ["name", "price"]
5  }
```

更加适合生产环境的使用, 可以构建复杂的查询

### 1基本实例

GET /\_search { "query": { "match\_all": {} } }

### 2、Query DSL的基本语法

```
1 {
2 QUERY_NAME: {
```

```
ARGUMENT: VALUE,
 4
           ARGUMENT: VALUE,...
       }
 5
   }
6
7
8
   {
9
     QUERY_NAME: {
          FIELD_NAME: {
10
11
              ARGUMENT: VALUE,
              ARGUMENT: VALUE, ...
12
13
          }
       }
14
15 }
```

#### 示例:

```
1   GET /test_index/test_type/_search
2   {
3     "query": {
4         "match": {
5             "test_field": "test"
6         }
7     }
8   }
```

### 3、如何组合多个搜索条件

搜索需求: title必须包含elasticsearch,content可以包含elasticsearch也可以不包含,author\_id必 须不为111

```
1
 2
     "took": 1,
      "timed_out": false,
 3
     "_shards": {
 4
 5
        "total": 5,
         "successful": 5,
 6
         "failed": 0
 7
8
      },
     "hits": {
9
         "total": 3,
10
11
         "max_score": 1,
12
         "hits": [
```

```
13
14
              "_index": "website",
              "_type": "article",
15
              "_id": "2",
16
              "_score": 1,
17
18
              " source": {
                "title": "my hadoop article",
19
                "content": "hadoop is very bad",
20
21
               "author_id": 111
22
             }
23
           },
24
              " index": "website",
25
26
              "_type": "article",
              " id": "1",
27
              "_score": 1,
28
              "_source": {
29
30
                "title": "my elasticsearch article",
31
                "content": "es is very bad",
               "author id": 110
32
33
             }
34
           },
35
              "_index": "website",
36
37
             "_type": "article",
38
              "_id": "3",
              " score": 1,
39
              "_source": {
40
                "title": "my elasticsearch article",
41
42
                "content": "es is very goods",
               "author_id": 111
43
44
             }
45
           }
46
         ]
47
      }
48
    }
49
50
    GET /website/article/_search
51
52
      "query": {
53
         "bool": {
54
            "must": [
55
56
                "match": {
                 "title": "elasticsearch"
57
58
               }
59
             }
60
            ],
            "should": [
61
```

```
62
                "match": {
63
                  "content": "elasticsearch"
64
65
66
              }
67
            ],
            "must_not": [
68
69
             {
70
                "match": {
                  "author id": 111
71
72
73
              }
74
            ]
75
          }
76
      }
77
    }
78
79
    GET /test_index/_search
80
          "query": {
81
82
                  "bool": {
                      "must": { "match": { "name": "tom" }},
83
84
                      "should": [
85
                           { "match":
                                            { "hired": true }},
86
                           { "bool": {
87
                               "must":
                                            { "match": { "personality":
    "good" }},
                               "must_not": { "match": { "rude": true }}
88
89
                           }}
90
                      ],
                      "minimum_should_match": 1
91
92
                  }
93
         }
94
    }
95
```

# 3、query filter

搜索商品名称包含yagao,而且售价大于25元的商品

```
7
                          "name" : "yagao"
 8
                     }
 9
                 },
                 "filter" : {
10
11
                     "range" : {
12
                         "price" : { "gt" : 25 }
13
14
                 }
15
           }
16
17 }
```

# 4、full-text search(全文检索)

producer这个字段,会先被拆解,建立倒排索引 special 4 yagao 4 producer 1,2,3,4 gaolujie 1 zhognhua 3 jiajieshi 2 yagao producer ---> yagao和producer

```
1
    {
 2
      "took": 4,
      "timed_out": false,
 3
 4
      " shards": {
 5
         "total": 5,
         "successful": 5,
 6
 7
         "failed": 0
8
      },
 9
      "hits": {
10
         "total": 4,
         "max score": 0.70293105,
11
         "hits": [
12
13
              "_index": "ecommerce",
14
              "_type": "product",
15
              "_id": "4",
16
              "_score": 0.70293105,
17
              " source": {
18
                "name": "special yagao",
19
```

```
"desc": "special meibai",
20
21
                "price": 50,
                "producer": "special yagao producer",
22
                "tags": [
23
                  "meibai"
24
25
                ]
26
             }
27
            },
28
              " index": "ecommerce",
29
              "_type": "product",
30
              "_id": "1",
31
              " score": 0.25811607,
32
33
              "_source": {
                "name": "gaolujie yagao",
34
                "desc": "gaoxiao meibai",
35
                "price": 30,
36
37
                "producer": "gaolujie producer",
38
                "tags": [
                  "meibai",
39
40
                  "fangzhu"
41
                1
42
             }
            },
43
44
           {
              "_index": "ecommerce",
45
              "_type": "product",
46
              " id": "3",
47
              "_score": 0.25811607,
48
49
              " source": {
                "name": "zhonghua yagao",
50
                "desc": "caoben zhiwu",
51
                "price": 40,
52
                "producer": "zhonghua producer",
53
                "tags": [
54
                  "qingxin"
55
56
                ]
              }
57
58
            },
59
              " index": "ecommerce",
60
61
              "_type": "product",
              "_id": "2",
62
              " score": 0.1805489,
63
              " source": {
64
                "name": "jiajieshi yagao",
65
                "desc": "youxiao fangzhu",
66
                "price": 25,
67
                "producer": "jiajieshi producer",
68
```

```
69 "tags": [
70 "fangzhu"

71 ]
72 }
73 }
74 ]
75 }
76 }
```

# 5、phrase search(短语搜索)

跟全文检索相对应,相反,全文检索会将输入的搜索串拆解开来,去倒排索引里面去一一匹配,只要 能匹配上任意一个拆解后的单词,就可以作为结果返回 phrase search,要求输入的搜索串,必须在 指定的字段文本中,完全包含一模一样的,才可以算匹配,才能作为结果返回

```
1
    GET /ecommerce/product/ search
2
 3
        "query" : {
             "match_phrase" : {
                 "producer" : "yagao producer"
 5
 6
 7
        }
8
    }
9
10
      "took": 11,
11
      "timed_out": false,
12
      "_shards": {
13
        "total": 5,
14
        "successful": 5,
15
        "failed": 0
16
17
      },
      "hits": {
18
         "total": 1,
19
          "max_score": 0.70293105,
20
21
          "hits": [
22
23
              " index": "ecommerce",
2.4
              "_type": "product",
25
              " id": "4",
26
              "_score": 0.70293105,
              " source": {
27
                "name": "special yagao",
28
                "desc": "special meibai",
29
                "price": 50,
30
                "producer": "special yagao producer",
31
                "tags": [
32
33
                  "meibai"
```

# 6、highlight search(高亮搜索结果)

```
1
    GET /ecommerce/product/_search
 2
         "query" : {
 3
 4
            "match" : {
 5
                 "producer" : "producer"
 6
             }
7
        },
         "highlight": {
             "fields" : {
9
                 "producer" : {}
10
11
             }
         }
12
13 }
```

# 聚合索引

# 第一个分析需求:计算每个tag下的商品数量

```
1   GET /ecommerce/product/_search
2   {
3     "aggs": {
4         "group_by_tags": {
5             "terms": { "field": "tags" }
6         }
7     }
8   }
```

将文本field的fielddata属性设置为true

```
1
   PUT /ecommerce/_mapping/product
2
3
    "properties": {
       "tags": {
4
5
         "type": "text",
         "fielddata": true
6
7
       }
8
    }
9 }
```

```
1  GET /ecommerce/product/_search
2  {
3    "size": 0,
4    "aggs": {
5        "all_tags": {
6          "terms": { "field": "tags" }
7        }
8     }
9  }
```

```
1
2
      "took": 20,
 3
      "timed_out": false,
 4
      " shards": {
 5
         "total": 5,
         "successful": 5,
 6
7
         "failed": 0
8
      },
9
      "hits": {
        "total": 4,
10
         "max_score": 0,
11
         "hits": []
12
13
      },
      "aggregations": {
14
         "group_by_tags": {
15
           "doc_count_error_upper_bound": 0,
16
           "sum_other_doc_count": 0,
17
18
           "buckets": [
19
               "key": "fangzhu",
20
21
              "doc_count": 2
             },
22
23
               "key": "meibai",
24
              "doc_count": 2
25
26
             },
27
28
               "key": "qingxin",
```

# 第二个聚合分析的需求:对名称中包含yagao的商品,计算每个tag下的商品数量

```
GET /ecommerce/product/ search
 2
 3
     "size": 0,
 4
     "query": {
 5
         "match": {
          "name": "yagao"
 6
 7
        }
     },
 8
     "aggs": {
9
        "all tags": {
10
           "terms": {
11
            "field": "tags"
12
13
          }
14
         }
15
     }
16 }
```

# 第三个聚合分析的需求:先分组,再算每组的平均值,计算每个 tag下的商品的平均价格

```
GET /ecommerce/product/_search
1
 2
 3
         "size": 0,
         "aggs" : {
 4
 5
             "group_by_tags" : {
                  "terms" : { "field" : "tags" },
 6
 7
                  "aggs" : {
 8
                      "avg_price" : {
                          "avg" : { "field" : "price" }
 9
10
                      }
                  }
11
12
             }
13
         }
   }
14
```

```
1
    {
 2
      "took": 8,
      "timed_out": false,
 3
      "_shards": {
 4
 5
         "total": 5,
         "successful": 5,
 6
 7
         "failed": 0
 8
      },
      "hits": {
9
10
         "total": 4,
         "max score": 0,
11
12
         "hits": []
13
      },
14
      "aggregations": {
15
         "group_by_tags": {
            "doc_count_error_upper_bound": 0,
16
            "sum_other_doc_count": 0,
17
            "buckets": [
18
19
20
                "key": "fangzhu",
                "doc_count": 2,
21
                "avg_price": {
22
                 "value": 27.5
23
24
                }
25
              },
26
             {
27
                "key": "meibai",
                "doc count": 2,
28
                "avg_price": {
29
                 "value": 40
30
31
                }
             },
32
33
                "key": "qingxin",
34
               "doc_count": 1,
35
                "avg_price": {
36
                  "value": 40
37
38
                }
39
40
            ]
41
         }
42
43 }
```

第四个数据分析需求: 计算每个tag下的商品的平均价格,并且按照平均价格降序排序

```
GET /ecommerce/product/_search
 2
    {
         "size": 0,
3
         "aggs" : {
 4
 5
             "all_tags" : {
                 "terms" : { "field" : "tags", "order": { "avg_price":
 6
    "desc" } },
                 "aggs" : {
7
8
                     "avg price" : {
                         "avg" : { "field" : "price" }
9
10
11
                 }
12
            }
13
        }
14 }
```

# 第五个数据分析需求:按照指定的价格范围区间进行分组,然后在每组内再按照tag进行分组,最后再计算每组的平均价格

```
GET /ecommerce/product/_search
1
2
 3
      "size": 0,
 4
      "aggs": {
         "group_by_price": {
           "range": {
 6
             "field": "price",
              "ranges": [
8
9
                  "from": 0,
10
                 "to": 20
11
12
                },
13
                  "from": 20,
14
                 "to": 40
15
16
                },
17
                  "from": 40,
18
                  "to": 50
19
20
                }
21
22
           },
            "aggs": {
23
24
              "group_by_tags": {
                "terms": {
25
                 "field": "tags"
26
27
                },
                "aggs": {
28
```

```
29
                    "average_price": {
30
                      "avg": {
                         "field": "price"
31
32
33
                    }
34
                 }
35
               }
36
             }
37
          }
38
39
    }
```

# document 操作

## 1、document的全量替换

(1) 语法与创建文档是一样的,如果document id不存在,那么就是创建;如果document id已经存在,那么就是全量替换操作,替换document的json串内容(2)document是不可变的,如果要修改document的内容,第一种方式就是全量替换,直接对document重新建立索引,替换里面所有的内容(3)es会将老的document标记为deleted,然后新增我们给定的一个document,当我们创建越来越多的document的时候,es会在适当的时机在后台自动删除标记为deleted的document

# 2、document的强制创建

(1)创建文档与全量替换的语法是一样的,有时我们只是想新建文档,不想替换文档,如果强制进行创建呢? (2)PUT /index/type/id?op\_type=create,PUT /index/type/id/\_create

### 3、document的删除

(1) DELETE /index/type/id (2) 不会理解物理删除,只会将其标记为deleted,当数据越来越多的时候,在后台自动删除

# ES 的脚本

es,其实是有个内置的脚本支持的,可以基于groovy脚本实现各种各样的复杂操作基于groovy脚本. 数据准备:

```
1  PUT /test_index/test_type/11
2  {
3     "num": 0,
4     "tags": []
5  }
```

# (1) 内置脚本

```
1
  POST /test_index/test_type/11/_update
2
      "script" : "ctx._source.num+=1"
 3
 4
 5
 6
7
     "_index": "test_index",
     "_type": "test_type",
8
     "_id": "11",
     "_version": 2,
10
     "found": true,
11
     "_source": {
12
        "num": 1,
13
14
        "tags": []
15
     }
16 }
```

### (2) 外部脚本

ctx.\_source.tags+=new\_tag

```
POST /test_index/test_type/11/_update
2
    "script": {
 3
       "lang": "groovy",
4
       "file": "test-add-tags",
5
       "params": {
 6
          "new_tag": "tag1"
7
        }
9
    }
10 }
```

### (3) 用脚本删除文档

```
1
    ctx.op = ctx._source.num == count ? 'delete' : 'none'
 2
 3
    POST /test_index/test_type/11/_update
 4
 5
      "script": {
         "lang": "groovy",
 6
         "file": "test-delete-document",
 8
         "params": {
           "count": 1
 9
10
11
12
    }
```

# (4) upsert操作

```
1   POST /test_index/test_type/11/_update
2   {
3     "doc": {
4          "num": 1
5     }
6   }
```

返回:

```
1
    {
 2
      "error": {
          "root_cause": [
 3
 4
              "type": "document missing exception",
 5
 6
              "reason": "[test_type][11]: document missing",
 7
              "index_uuid": "6m0G7yx7R1KECWWGnfH1sw",
 8
              "shard": "4",
 9
              "index": "test_index"
10
           }
11
         ],
         "type": "document missing exception",
12
          "reason": "[test_type][11]: document missing",
13
         "index_uuid": "6m0G7yx7R1KECWWGnfH1sw",
14
         "shard": "4",
15
          "index": "test_index"
16
17
      "status": 404
18
19
    }
```

如果指定的document不存在,就执行upsert中的初始化操作;如果指定的document存在,就执行doc或者script指定的partial update操作

```
POST /test_index/test_type/11/_update
{
    "script" : "ctx._source.num+=1",
    "upsert": {
        "num": 0,
        "tags": []
}
}
```

partial update 自动执行乐观锁控制.

使用参数retry 策略进行重试.

# 批量命令

#### 1、批量查询的好处

就是一条一条的查询,比如说要查询100条数据,那么就要发送100次网络请求,这个开销还是很大的如果进行批量查询的话,查询100条数据,就只要发送1次网络请求,网络请求的性能开销缩减100倍

- 2、mget的语法
- (1) 一条一条的查询

GET /test\_index/test\_type/1 GET /test\_index/test\_type/2

(2) mget批量查询

```
1 GET /_mget
2
       "docs" : [
 3
 4
          {
 5
             " index" : "test index",
              "_type" : "test_type",
 6
             " id" :
 7
8
           },
9
              "_index" : "test_index",
10
             "_type" : "test_type",
11
              " id" :
12
         }
13
14
       ]
15 }
```

```
"_type": "test_type",
           "_id": "1",
 6
 7
           "_version": 2,
           "found": true,
8
9
           "_source": {
10
            "test_field1": "test field1",
            "test field2": "test field2"
11
12
          }
13
        },
14
          "_index": "test_index",
15
          "_type": "test_type",
16
          "_id": "2",
17
           "_version": 1,
18
          "found": true,
19
          "_source": {
20
           "test content": "my test"
21
22
          }
23
         }
24
     ]
25 }
```

(3) 如果查询的document是一个index下的不同type种的话

```
GET /test_index/_mget
2
3
     "docs" : [
4
         {
5
             "_type" : "test_type",
            "_id" : 1
6
7
         },
8
            "_type" : "test_type",
9
             "_id" : 2
10
11
12
      ]
13 }
```

(4) 如果查询的数据都在同一个index下的同一个type下,最简单了

```
1   GET /test_index/test_type/_mget
2   {
3     "ids": [1, 2]
4   }
```

3、mget的重要性

可以说mget是很重要的,一般来说,在进行查询的时候,如果一次性要查询多条数据的话,那么一定要用batch批量操作的api 尽可能减少网络开销次数,可能可以将性能提升数倍,甚至数十倍,非常非常之重要

# 批量增删改

### 1、bulk语法

```
post /_bulk

delete": { "_index": "test_index", "_type": "test_type", "_id": "3" }}

function "state": { "_index": "test_index", "_type": "test_type", "_id": "12" }}

function "state": { "_index": "test12" }

function "state": { "_index": "test_index", "_type": "test_type", "_id": "2" }}

function "state": { "_index": "test_index", "_type": "test_type", "_id": "1", "_retry_on_conflict": 3} }

function "state": { "_index": "test_index", "_type": "test_type", "_id": "1", "_retry_on_conflict": 3} }

function "state" "state = "
```

#### 每一个操作要两个json串,语法如下:

```
1 {"action": {"metadata"}}
2 {"data"}
```

举例,比如你现在要创建一个文档,放bulk里面,看起来会是这样子的:

```
1 {"index": {"_index": "test_index", "_type", "test_type", "_id": "1"}}
2 {"test_field1": "test1", "test_field2": "test2"}
```

#### 有哪些类型的操作可以执行呢?

- (1) delete: 删除一个文档,只要1个json串就可以了
- (2) create: PUT /index/type/id/\_create, 强制创建
- (3) index: 普通的put操作, 可以是创建文档, 也可以是全量替换文档
- (4) update: 执行的partial update操作

bulk api对json的语法,有严格的要求,每个json串不能换行,只能放一行,同时一个json串和一个 json串之间,必须有一个换行

```
"reason": "Unexpected end-of-input: expected close marker for
    Object (start marker at [Source:
    org.elasticsearch.transport.netty4.ByteBufStreamInput@5a5932cd; line:
    1, column: 1])\n at [Source:
    org.elasticsearch.transport.netty4.ByteBufStreamInput@5a5932cd; line:
    1, column: 3]"
 7
          }
 8
        1,
 9
        "type": "json_e_o_f_exception",
        "reason": "Unexpected end-of-input: expected close marker for
10
    Object (start marker at [Source:
    org.elasticsearch.transport.netty4.ByteBufStreamInput@5a5932cd; line:
    1, column: 1])\n at [Source:
    org.elasticsearch.transport.netty4.ByteBufStreamInput@5a5932cd; line:
    1, column: 3]"
     },
11
     "status": 500
12
13
   }
14
15
      "took": 41,
16
17
      "errors": true,
18
      "items": [
19
          "delete": {
20
            "found": true,
21
            " index": "test index",
22
            "_type": "test_type",
23
            "_id": "10",
24
            " version": 3,
25
            "result": "deleted",
26
27
            "_shards": {
              "total": 2,
2.8
              "successful": 1,
29
              "failed": 0
30
31
            },
            "status": 200
32
33
          }
34
        },
35
          "create": {
36
37
            " index": "test_index",
            "_type": "test_type",
38
            " id": "3",
39
            " version": 1,
40
            "result": "created",
41
42
            "_shards": {
              "total": 2,
43
44
              "successful": 1,
```

```
"failed": 0
45
46
            },
             "created": true,
47
            "status": 201
48
49
          }
50
        },
51
          "create": {
52
53
             "_index": "test_index",
             " type": "test type",
54
             " id": "2",
55
            "status": 409,
56
57
             "error": {
               "type": "version_conflict_engine_exception",
58
              "reason": "[test_type][2]: version conflict, document
59
    already exists (current version [1])",
              "index uuid": "6m0G7yx7R1KECWWGnfH1sw",
60
              "shard": "2",
61
              "index": "test_index"
62
63
            }
64
          }
65
        },
66
        {
          "index": {
67
            " index": "test_index",
68
             "_type": "test_type",
69
             " id": "4",
70
71
             "_version": 1,
            "result": "created",
72
             " shards": {
73
              "total": 2,
74
75
              "successful": 1,
              "failed": 0
76
77
            },
            "created": true,
78
             "status": 201
79
80
         }
81
        },
82
        {
83
          "index": {
             " index": "test index",
84
85
             "_type": "test_type",
             "_id": "2",
86
             "_version": 2,
87
             "result": "updated",
88
             " shards": {
89
90
               "total": 2,
              "successful": 1,
91
              "failed": 0
92
```

```
93
             },
 94
             "created": false,
             "status": 200
 95
 96
          }
 97
         },
 98
         {
99
          "update": {
             " index": "test_index",
100
101
             "_type": "test_type",
             " id": "1",
102
             " version": 3,
103
             "result": "updated",
104
105
             " shards": {
              "total": 2,
106
              "successful": 1,
107
              "failed": 0
108
109
             },
             "status": 200
110
111
           }
112
       }
      ]
113
114 }
```

bulk操作中,任意一个操作失败,是不会影响其他的操作的,但是在返回结果里,会告诉你异常日志

```
1 POST /test index/ bulk
 2 { "delete": { "_type": "test_type", "_id": "3" }}
 3 { "create": { " type": "test type", " id": "12" }}
 4 { "test_field": "test12" }
   { "index": { "_type": "test_type" }}
   { "test field": "auto-generate id test" }
 6
   { "index": { "_type": "test_type", "_id": "2" }}
 7
   { "test field": "replaced test2" }
  { "update": { "_type": "test_type", "_id": "1", "_retry_on_conflict" :
10 { "doc" : {"test field2" : "bulk test1"} }
11
12 POST /test_index/test_type/_bulk
13 { "delete": { " id": "3" }}
14 { "create": { "_id": "12" }}
15 { "test_field":
                     "test12" }
16 { "index": { }}
17 { "test_field":
                     "auto-generate id test" }
   { "index": { "_id": "2" }}
18
   { "test field": "replaced test2" }
19
20 { "update": { "_id": "1", "_retry_on_conflict" : 3} }
    { "doc" : {"test field2" : "bulk test1"} }
```

### 2、bulk size最佳大小

bulk request会加载到内存里,如果太大的话,性能反而会下降,因此需要反复尝试一个最佳的bulk size。一般从1000~5000条数据开始,尝试逐渐增加。另外,如果看大小的话,最好是在5~15MB之间

### 写一致性原理

### 写操作时的consistency参数

我们在发送任何一个增删改操作的时候,比如说put /index/type/id,都可以带上一个consistency参数,指明我们想要的写一致性是什么? put /index/type/id?consistency=quorum

one:要求我们这个写操作,只要有一个primary shard是active活跃可用的,就可以执行 all:要求我们这个写操作,必须所有的primary shard和replica shard都是活跃的,才可以执行这个写操作quorum:默认的值,要求所有的shard中,必须是大部分的shard都是活跃的,可用的,才可以执行这个写操作

### quorum机制

写之前必须确保大多数shard都可用,int( (primary + number\_of\_replicas) / 2 ) + 1,当 number\_of\_replicas>1时才生效

quroum = int( (primary + number\_of\_replicas) / 2 ) + 1 举个例子,3个primary shard,number\_of\_replicas=1,总共有3 + 3 \* 1 = 6个shard quorum = int( (3 + 1) / 2 ) + 1 = 3 所以,要求6个shard中至少有3个shard是active状态的,才可以执行这个写操作

#### 例外:

● 如果节点数少于quorum数量,可能导致quorum不齐全,进而导致无法执行任何写操作

3个primary shard, replica=1,要求至少3个shard是active, 3个shard按照之前学习的 shard&replica机制,必须在不同的节点上,如果说只有2台机器的话,是不是有可能出现说,3个 shard都没法分配齐全,此时就可能会出现写操作无法执行的情况

es提供了一种特殊的处理场景,就是说当number\_of\_replicas>1时才生效,因为假如说,你就一个primary shard,replica=1,此时就2个shard

(1 + 1 / 2) + 1 = 2,要求必须有2个shard是活跃的,但是可能就1个node,此时就1个shard是活跃的,如果你不特殊处理的话,导致我们的单节点集群就无法工作

• quorum不齐全时, wait, 默认1分钟, timeout, 100, 30s

等待期间,期望活跃的shard数量可以增加,最后实在不行,就会timeout 我们其实可以在写操作的时候,加一个timeout参数,比如说put /index/type/id?timeout=30,这个就是说自己去设定 quorum不齐全的时候,es的timeout时长,可以缩短,也可以增长

# bulk api奇特的json格式原因

```
1 {"action": {"meta"}}\n
   {"data"}\n
 3 {"action": {"meta"}}\n
 4 {"data"}\n
 6
   [ {
 7
      "action": {
 8
9
     "data": {
10
11
12
      }
13 }]
```

- 1、bulk中的每个操作都可能要转发到不同的node的shard去执行
- 2、如果采用比较良好的json数组格式

允许任意的换行,整个可读性非常棒,读起来很爽,es拿到那种标准格式的json串以后,要按照下述流程去进行处理

- (1) 将json数组解析为JSONArray对象,这个时候,整个数据,就会在内存中出现一份一模一样的拷贝,一份数据是json文本,一份数据是JSONArray对象 (2) 解析json数组里的每个json,对每个请求中的document进行路由 (3) 为路由到同一个shard上的多个请求,创建一个请求数组 (4) 将这个请求数组序列化 (5) 将序列化后的请求数组发送到对应的节点上去
- 3、耗费更多内存,更多的jvm gc开销

我们之前提到过bulk size最佳大小的那个问题,一般建议说在几千条那样,然后大小在10MB左右,所以说,可怕的事情来了。假设说现在100个bulk请求发送到了一个节点上去,然后每个请求是10MB,100个请求,就是1000MB = 1GB,然后每个请求的json都copy一份为jsonarray对象,此时内存中的占用就会翻倍,就会占用2GB的内存,甚至还不止。因为弄成jsonarray之后,还可能会多搞一些其他的数据结构,2GB+的内存占用。

占用更多的内存可能就会积压其他请求的内存使用量,比如说最重要的搜索请求,分析请求,等等,此时就可能会导致其他请求的性能急速下降 另外的话,占用内存更多,就会导致java虚拟机的垃圾回收次数更多,跟频繁,每次要回收的垃圾对象更多,耗费的时间更多,导致es的java虚拟机停止工作线程的时间更多

#### 4、现在的奇特格式

 ${\color=0.05cm} {\color=0.05cm} {\color=0.05$ 

(1) 不用将其转换为json对象,不会出现内存中的相同数据的拷贝,直接按照换行符切割json(2)对每两个一组的json,读取meta,进行document路由(3)直接将对应的json发送到node上去

5、最大的优势在于,不需要将json数组解析为一个JSONArray对象,形成一份大数据的拷贝,浪费内存空间,尽可能地保证性能

### 搜索

```
GET / search
 2
 3
 4
     "took": 6,
 5
     "timed_out": false,
      " shards": {
 6
         "total": 6,
 7
         "successful": 6,
 8
         "failed": 0
9
10
     },
     "hits": {
11
12
         "total": 10,
13
         "max_score": 1,
14
         "hits": [
15
             " index": ".kibana",
16
              " type": "config",
17
              " id": "5.2.0",
18
19
              "_score": 1,
              "_source": {
20
                "buildNum": 14695
21
22
            }
23
          }
24
         ]
25
     }
26 }
```

#### 返回结果:

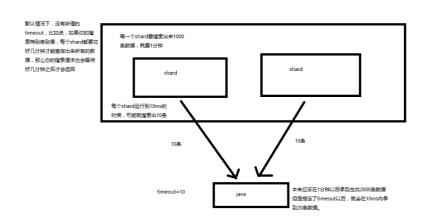
- took:整个搜索请求花费了多少毫秒
- hits.total: 本次搜索,返回了几条结果
- hits.max\_score: 本次搜索的所有结果中,最大的相关度分数是多少,每一条document对于 search的相关度,越相关,\_score分数越大,排位越靠前
- hits.hits: 默认查询前10条数据,完整数据,\_score降序排序
- shards: shards fail的条件(primary和replica全部挂掉),不影响其他shard。默认情况下来说,一个搜索请求,会打到一个index的所有primary shard上去,当然了,每个primary shard都可能会有一个或多个replic shard,所以请求也可以到primary shard的其中一个replica shard上去。
- timeout: 默认无timeout, latency平衡completeness, 手动指定timeout, timeout查询执行机制

timeout=10ms, timeout=1s, timeout=1m GET /\_search?timeout=10m

我们有些惶繁应用,对时间是很敏感的,比如说我么的电商网站,你不能 说让用户等各10分钟,才能等到一次 撞紊清末的结果,如果那样的话,人 家早就走了,不来买东西了。

timeout机侧,指定每个shard,就 只能在timeout时间范围内,将撞索 到的部分数据(也可能全都撞索到 了),直接理解返回给client程序, 而不是等到所有的数据全都搜索出来 以后高级同

确保说,一次搜索请求可以在用户指 定的timeout时长内完成。为一些时 间敏感的搜索应用提供良好的支持。



# 1、multi-index和multi-type搜索模式

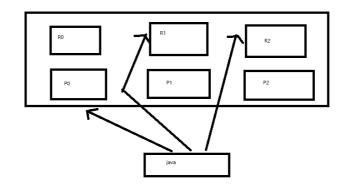
告诉你如何一次性搜索多个index和多个type下的数据

/search: 所有索引,所有type下的所有数据都搜索出来 /index1/search: 指定一个index,搜索其下所有type的数据 /index1,index2/search: 同时搜索两个index下的数据 /1,2/search: 按照通配符去匹配多个索引 /index1/type1/search: 搜索一个index下指定的type的数据 /index1/type2/search: 可以搜索一个index下多个type的数据 /index1,index2/type1,type2/search: 搜索多个index下的多个type的数据 /all/type1,type2/search: all,可以代表搜索所有index下的指定type的数据

### 2、初步图解一下简单的搜索原理

client发送一个搜索请求,会把请求打到所有的 primary shard上去执 行,因为每个shard都包 含部分数据,所以每个 shard上都可能会包含搜 索请求的结果

但是如果primary shard 有replica shard,那么请 求也可以打到replica shard上去



### 分页搜索

● 1、讲解如何使用es进行分页搜索的语法

size, from

GET /search?size=10 GET /search?size=10&from=0 GET /\_search?size=10&from=20 分页的上机实验

```
1  GET /test_index/test_type/_search
2  
3  "hits": {
4     "total": 9,
5     "max_score": 1,
```

#### 我们假设将这9条数据分成3页,每一页是3条数据,来实验一下这个分页搜索的效果

```
GET /test_index/test_type/_search?from=0&size=3
 2
 3
    {
      "took": 2,
 4
 5
     "timed out": false,
     " shards": {
 6
7
         "total": 5,
 8
        "successful": 5,
9
         "failed": 0
10
     },
     "hits": {
11
         "total": 9,
12
13
         "max score": 1,
         "hits": [
14
15
             " index": "test index",
16
             " type": "test type",
17
             " id": "8",
18
             "_score": 1,
19
20
             " source": {
               "test_field": "test client 2"
21
22
23
           },
24
           {
25
              "_index": "test_index",
             " type": "test type",
26
27
             " id": "6",
             " score": 1,
28
             "_source": {
29
               "test_field": "tes test"
30
31
             }
32
           },
33
34
             " index": "test index",
              "_type": "test_type",
35
             " id": "4",
36
37
             "_score": 1,
              "_source": {
38
39
               "test_field": "test4"
```

第一页: id=8,6,4

GET /test\_index/test\_type/\_search?from=3&size=3

第二页: id=2,自动生成,7

GET /test\_index/test\_type/\_search?from=6&size=3

第三页: id=1,11,3

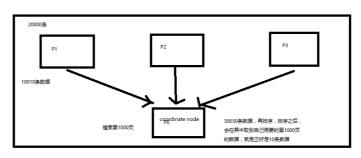
2、什么是deep paging问题?为什么会产生这个问题,它的底层原理是什么?

什么叫deep paging?簡單来说,就是撞索的特別深,比如总共有6000多数据,每个shard上分了2000多数据。 每页是10条数据,这个时候,你要撞索到第1000页,实际 上要拿到的是10001~10010大家自己思考一下,是第几条 到第几条数啊?

每个shard,其实都要返回的是最后10条数据,不是这样理解。。。。。错误!!!!

你的请求首先可能是打到一个不包含这个index的shard的 node上去,这个node就是一个coordinate node,那么这个coordinate node就会将搜索请求转发到index的三个shard所在的node上去。

比如说我们刚才说的情况下,要搜索6000多数据中的第
1000页,实际上每个shard都要将内部的2000多数据中的
第10001~10010条数据,拿出来,不是才10条,是10010
条数据,3个shard每个shard都返回10010条数据给
coordinate node,coordinate node会收到应共30330条数据,然后在这些影響中进行排序,5core,相关度分
数,然后取到排位是高的前10条数据,其实就是统门要约



擅棄的过深的时候,就需要在coordinate node上保存大量的数据,还要进行大量数据的排序,排序之后,再取出对应的部一页。 所以这个过程,即转费网络带竞,转费内存,还转费cpu。 所以deep paging的性能问题。 我们应该尽量提免出现这种 deep paging操作。

## 1、query string基础语法

```
GET /test_index/test_type/_search?q=test_field:test
GET /test_index/test_type/_search?q=+test_field:test
GET /test_index/test_type/_search?q=-test_field:test
```

一个是掌握g=field:search content的语法,还有一个是掌握+和-的含义

### 2、\_all metadata的原理和作用

```
1 GET /test_index/test_type/_search?q=test
2
```

直接可以搜索所有的field,任意一个field包含指定的关键字就可以搜索出来。我们在进行中搜索的时候,难道是对document中的每一个field都进行一次搜索吗?不是的

es中的all元数据,在建立索引的时候,我们插入一条document,它里面包含了多个field,此时,es会自动将多个field的值,全部用字符串的方式串联起来,变成一个长的字符串,作为all field的值,同时建立索引

后面如果在搜索的时候,没有对某个field指定搜索,就默认搜索\_all field,其中是包含了所有field的 值的

举个例子

```
1 {
2    "name": "jack",
3    "age": 26,
4    "email": "jack@sina.com",
5    "address": "guamgzhou"
6 }
```

"jack 26 j<u>ack@sina.com</u> guangzhou",作为这一条document的\_all field的值,同时进行分词后建立 对应的倒排索引

生产环境不使用

# mapping

插入几条数据,让es自动为我们建立一个索引

```
1 PUT /website/article/1
 2 {
     "post date": "2017-01-01",
 3
     "title": "my first article",
     "content": "this is my first article in this website",
 5
     "author id": 11400
 7
   }
 8
   PUT /website/article/2
9
10
11
     "post_date": "2017-01-02",
12
     "title": "my second article",
13
     "content": "this is my second article in this website",
     "author id": 11400
14
15
16
17
   PUT /website/article/3
18
19
      "post_date": "2017-01-03",
      "title": "my third article",
20
      "content": "this is my third article in this website",
21
      "author id": 11400
22
23 }
```

#### 尝试各种搜索

GET /website/article/search?q=2017 3条结果

GET /website/article/search?q=2017-01-01 3条结果 GET /website/article/search?q=post\_date:2017-01-01 1条结果 GET /website/article/search?q=post\_date:2017 1条结果

查看es自动建立的mapping,带出什么是mapping的知识点 自动或手动为index中的type建立的一种数据结构和相关配置,简称为mapping dynamic mapping,自动为我们建立index,创建type,以及type对应的mapping,mapping中包含了每个field对应的数据类型,以及如何分词等设置 我们当然,后面会讲解,也可以手动在创建数据之前,先创建index和type,以及type对应的mapping

```
GET /website/_mapping/article
 2
 3
      "website": {
 4
          "mappings": {
 5
            "article": {
 6
              "properties": {
                 "author id": {
 8
                   "type": "long"
 9
10
                },
                "content": {
11
                   "type": "text",
12
                   "fields": {
13
                     "keyword": {
14
                       "type": "keyword",
15
                       "ignore above": 256
16
17
                     }
18
                   }
19
                },
                 "post_date": {
20
                   "type": "date"
21
22
                "title": {
23
                   "type": "text",
24
                   "fields": {
25
26
                     "keyword": {
                       "type": "keyword",
27
                       "ignore_above": 256
28
29
                     }
30
                   }
31
                }
32
              }
33
            }
34
          }
35
36
    }
37
```

搜索结果为什么不一致,因为es自动建立mapping的时候,设置了不同的field不同的data type。不同的data type的分词、搜索等行为是不一样的。所以出现了\_all field和post\_date field的搜索表现完全不一样。

### 分词

doc1: I really liked my small dogs, and I think my mom also liked them. doc2: He never liked any dogs, so I hope that my mom will not expect me to liked him.

分词,初步的倒排索引的建立

word doc1 doc2

I \* \* really \* liked \* \* my \* \* small \* dogs \* and \* think \* mom \* \* also \* them \* He \* never \* any \* so \* hope \* that \* will \* not \* expect \* me \* to \* him \*

演示了一下倒排索引最简单的建立的一个过程

#### 搜索

mother like little dog,不可能有任何结果

mother like little dog

这个是不是我们想要的搜索结果???绝对不是,因为在我们看来,mother和mom有区别吗?同义词,都是妈妈的意思。like和liked有区别吗?没有,都是喜欢的意思,只不过一个是现在时,一个是过去时。little和small有区别吗?同义词,都是小小的。dog和dogs有区别吗?狗,只不过一个是单数,一个是复数。

normalization,建立倒排索引的时候,会执行一个操作,也就是说对拆分出的各个单词进行相应的处理,以提升后面搜索的时候能够搜索到相关联的文档的概率

时态的转换,单复数的转换,同义词的转换,大小写的转换

mom —> mother liked —> like small —> little dogs —> dog

重新建立倒排索引,加入normalization,再次用mother liked little dog搜索,就可以搜索到了

word doc1 doc2

I \* \* really \* like \* \* liked --> like my \* \* little \* small --> little dog \* \* dogs --> dog and \* think \* mom \* \* also \* them \* He \* never \* any \* so \* hope \* that \* will \* not \* expect \* me \* to \* him \*

mother like little dog,分词,normalization

mother --> mom like --> like little --> little dog --> dog

doc1和doc2都会搜索出来

doc1: I really liked my small dogs, and I think my mom also liked them. doc2: He never liked any dogs, so I hope that my mom will not expect me to liked him.

#### 1、什么是分词器

切分词语, normalization (提升recall召回率)

给你一段句子,然后将这段句子拆分成一个一个的单个的单词,同时对每个单词进行 normalization(时态转换,单复数转换),分瓷器 recall,召回率:搜索的时候,增加能够搜索到的 结果的数量

character filter: 在一段文本进行分词之前,先进行预处理,比如说最常见的就是,过滤html标签 (hello --> hello) ,& --> and (l&you --> l and you) tokenizer: 分词,hello you and me --> hello, you, and, me token filter: lowercase, stop word, synonymom, dogs --> dog, liked --> like, Tom --> tom, a/the/an --> 干掉,mother --> mom, small --> little

一个分词器,很重要,将一段文本进行各种处理,最后处理好的结果才会拿去建立倒排索引

#### 2、内置分词器的介绍

Set the shape to semi-transparent by calling set\_trans(5)

standard analyzer: set, the, shape, to, semi, transparent, by, calling, set\_trans, 5(默认的是 standard) simple analyzer: set, the, shape, to, semi, transparent, by, calling, set, trans whitespace analyzer: Set, the, shape, to, semi-transparent, by, calling, set\_trans(5) language analyzer(特定的语言的分词器,比如说,english,英语分词器): set, shape, semi, transpar, call, set\_tran, 5

# 创建mapping

1、如何建立索引

analyzed not\_analyzed no

2、修改mapping

只能创建index时手动建立mapping,或者新增field mapping,但是不能update field mapping

```
1
   PUT /website
3
      "mappings": {
4
         "article": {
            "properties": {
6
7
              "author id": {
                "type": "long"
8
9
              },
              "title": {
10
                "type": "text",
11
                "analyzer": "english"
12
13
              },
```

```
14
              "content": {
15
                "type": "text"
16
             },
              "post_date": {
17
                "type": "date"
18
19
             },
20
              "publisher_id": {
                "type": "text",
21
22
               "index": "not_analyzed"
23
           }
24
25
         }
26
      }
27
    }
28
    PUT /website
29
30
31
      "mappings": {
32
         "article": {
33
           "properties": {
34
              "author_id": {
               "type": "text"
35
36
             }
37
           }
38
         }
39
      }
40
41
42
43
      "error": {
         "root_cause": [
44
45
              "type": "index_already_exists_exception",
46
              "reason": "index [website/coldgJ-uTYGBEEOOL8GsQQ] already
47
    exists",
             "index_uuid": "coldgJ-uTYGBEEOOL8GsQQ",
48
49
             "index": "website"
           }
50
51
         ],
52
          "type": "index already exists exception",
53
          "reason": "index [website/coldgJ-uTYGBEEOOL8GsQQ] already exists",
54
         "index_uuid": "coldgJ-uTYGBEEOOL8GsQQ",
55
         "index": "website"
56
      },
      "status": 400
57
58
59
    PUT /website/ mapping/article
60
61
```

#### 3、测试mapping

```
GET /website/ analyze
 2
     "field": "content",
 3
    "text": "my-dogs"
 5
   }
   GET website/_analyze
7
8
9
     "field": "new_field",
     "text": "my dogs"
10
11
    }
12
13
14
      "error": {
         "root_cause": [
15
16
          {
17
             "type": "remote_transport_exception",
             "reason": "[4onsTYV][127.0.0.1:9300]
18
    [indices:admin/analyze[s]]"
19
          }
20
         ],
         "type": "illegal_argument_exception",
21
         "reason": "Can't process field [new_field], Analysis requests are
    only supported on tokenized fields"
23
     },
     "status": 400
24
25
   }
```

# mapping数据类型

#### 1, multivalue field

```
{ "tags": [ "tag1", "tag2" ]}
```

建立索引时与string是一样的,数据类型不能混

### 2 empty field

### 3. object field

```
1
    PUT /company/employee/1
 2
 3
      "address": {
 4
 5
        "country": "china",
        "province": "guangdong",
 6
        "city": "guangzhou"
 8
      },
 9
      "name": "jack",
10
      "age": 27,
      "join_date": "2017-01-01"
11
12
    }
13
    address: object类型
14
15
16
17
       "company": {
        "mappings": {
18
19
          "employee": {
             "properties": {
20
               "address": {
21
22
                 "properties": {
                   "city": {
23
24
                     "type": "text",
                     "fields": {
25
26
                       "keyword": {
                          "type": "keyword",
27
                         "ignore above": 256
28
29
                       }
                     }
30
31
                   },
                   "country": {
32
                     "type": "text",
33
                     "fields": {
34
                       "keyword": {
35
                         "type": "keyword",
36
                         "ignore_above": 256
37
38
                       }
                     }
39
40
                   },
                   "province": {
41
                     "type": "text",
42
43
                     "fields": {
                       "keyword": {
44
                          "type": "keyword",
45
```

```
46
                          "ignore_above": 256
                       }
47
48
                     }
                   }
49
50
                 }
51
               },
               "age": {
52
                "type": "long"
53
54
               },
               "join_date": {
55
                 "type": "date"
56
               },
57
               "name": {
58
59
                 "type": "text",
                 "fields": {
60
                   "keyword": {
61
                     "type": "keyword",
62
63
                     "ignore_above": 256
                   }
64
65
                 }
66
               }
67
            }
68
          }
69
        }
70
      }
71
    }
72
73
      "address": {
74
75
        "country": "china",
        "province": "guangdong",
76
        "city": "guangzhou"
77
78
      },
      "name": "jack",
79
      "age": 27,
80
      "join_date": "2017-01-01"
81
82
    }
83
84
85
        "name":
                            [jack],
        "age":
                         [27],
86
87
        "join_date":
                        [2017-01-01],
        "address.country":
88
89
        "address.province": [guangdong],
        "address.city": [guangzhou]
90
91
92
93
    {
        "authors": [
94
```

```
95
             { "age": 26, "name": "Jack White"},
 96
             { "age": 55, "name": "Tom Jones"},
             { "age": 39, "name": "Kitty Smith"}
 97
 98
         1
99
100
101
     底层存储:
102
    {
103
         "authors.age": [26, 55, 39],
         "authors.name": [jack, white, tom, jones, kitty, smith]
104
105 }
```

### search API

### 1、search api的基本语法

GET /search {}

GET /index1,index2/type1,type2/search {}

GET /\_search { "from": 0, "size": 10 }

## 2、http协议中get是否可以带上request body

HTTP协议,一般不允许get请求带上request body,但是因为get更加适合描述查询数据的操作,因此还是这么用了

GET / search?from=0&size=10

POST /\_search { "from":0, "size":10 }

碰巧,很多浏览器,或者是服务器,也都支持GET+request body模式

如果遇到不支持的场景,也可以用POST /\_search

#### 1、filter与query示例

```
PUT /company/employee/2 { "address": { "country": "china", "province": "jiangsu", "city": "nanjing" }, "name": "tom", "age": 30, "join_date": "2016-01-01" }
```

PUT /company/employee/3 { "address": { "country": "china", "province": "shanxi", "city": "xian" }, "name": "marry", "age": 35, "join\_date": "2015-01-01" }

搜索请求: 年龄必须大于等于30, 同时join\_date必须是2016-01-01

GET /company/employee/\_search { "query": { "bool": { "must": [ { "match": { "join\_date": "2016-01-01" } } ], "filter": { "range": { "age": { "gte": 30 } } } } }

2、filter与query对比大解密

filter,仅仅只是按照搜索条件过滤出需要的数据而已,不计算任何相关度分数,对相关度没有任何影响 query,会去计算每个document相对于搜索条件的相关度,并按照相关度进行排序

一般来说,如果你是在进行搜索,需要将最匹配搜索条件的数据先返回,那么用query;如果你只是要根据一些条件筛选出一部分数据,不关注其排序,那么用filter 除非是你的这些搜索条件,你希望越符合这些搜索条件的document越排在前面返回,那么这些搜索条件要放在query中;如果你不希望一些搜索条件来影响你的document排序,那么就放在filter中即可

#### 3、filter与query性能

filter,不需要计算相关度分数,不需要按照相关度分数进行排序,同时还有内置的自动cache最常使用filter的数据 query,相反,要计算相关度分数,按照分数进行排序,而且无法cache结果

### string sort

如果对一个string field进行排序,结果往往不准确,因为分词后是多个单词,再排序就不是我们想要的结果了 通常解决方案是,将一个string field建立两次索引,一个分词,用来进行搜索;一个不分词,用来进行排序

PUT /website { "mappings": { "article": { "properties": { "title": { "type": "text", "fields": { "raw": { "type": "string", "index": "not\_analyzed" } }, "fielddata": true }, "content": { "type": "text" }, "post\_date": { "type": "date" }, "author\_id": { "type": "long" } } } }

PUT /website/article/1 { "title": "first article", "content": "this is my second article", "post\_date": "2017-01-01", "author\_id": 110 }

{ "took": 2, "timed\_out": false, "shards": { "total": 5, "successful": 5, "failed": 0 }, "hits": { "total": 3, "max\_score": 1, "hits": [ { "index": "website", "type": "article", "id": "2", "score": 1, "source": { "title": "first article", "content": "this is my first article", "post\_date": "2017-02-01", "author\_id": 110 } }, { "index": "website", "type": "article", "id": "1", "score": 1, "source": { "title": "second article", "content": "this is my second article", "post\_date": "2017-01-01", "author\_id": 110 } }, { "index": "website", "type": "article", "id": "3", "score": 1, "source": { "title": "third article", "content": "this is my third article", "post\_date": "2017-03-01", "author\_id": 110 } } ] }

GET /website/article/\_search { "query": { "match\_all": {} }, "sort": [ { "title.raw": { "order": "desc" } } ]}

# dynamic mapping

### 1、定制dynamic策略

true:遇到陌生字段,就进行dynamic mapping false:遇到陌生字段,就忽略 strict:遇到陌生字段,就报错

PUT /my\_index { "mappings": { "my\_type": { "dynamic": "strict", "properties": { "title": { "type": "text" }, "address": { "type": "object", "dynamic": "true" } } } }

```
PUT /my_index/my_type/1 { "title": "my article", "content": "this is my article", "address": { "province": "guangdong", "city": "guangzhou" } }
```

{ "error": { "root\_cause": [ { "type": "strict\_dynamic\_mapping\_exception", "reason": "mapping set to strict, dynamic introduction of [content] within [my\_type] is not allowed" } ], "type": "strict\_dynamic\_mapping\_exception", "reason": "mapping set to strict, dynamic introduction of [content] within [my\_type] is not allowed" }, "status": 400 }

PUT /my\_index/my\_type/1 { "title": "my article", "address": { "province": "guangdong", "city": "guangzhou" } }

GET /my\_index/\_mapping/my\_type

```
{ "my_index": { "mappings": { "my_type": { "dynamic": "strict", "properties": { "address": { "dynamic": "true", "properties": { "city": { "type": "text", "fields": { "keyword": { "type": "keyword", "ignore_above": 256 } } }, "province": { "type": "text", "fields": { "keyword": { "type": "keyword", "ignore_above": 256 } } } }, "title": { "type": "text" } } } }
```

### 2、定制dynamic mapping策略

(1) date\_detection

默认会按照一定格式识别date,比如yyyy-MM-dd。但是如果某个field先过来一个2017-01-01的值,就会被自动dynamic mapping成date,后面如果再来一个"hello world"之类的值,就会报错。可以手动关闭某个type的date\_detection,如果有需要,自己手动指定某个field为date类型。

PUT /my\_index/\_mapping/my\_type { "date\_detection": false }

(2) 定制自己的dynamic mapping template(type level)

```
PUT /my_index { "mappings": { "my_type": { "dynamic_templates": [ { "en": { "match": "*_en", "match_mapping_type": "string", "mapping": { "type": "string", "analyzer": "english" } } ] }}
```

PUT /my\_index/my\_type/1 { "title": "this is my first article" }

PUT /my\_index/my\_type/2 { "title\_en": "this is my first article" }

title没有匹配到任何的dynamic模板,默认就是standard分词器,不会过滤停用词,is会进入倒排索引,用is来搜索是可以搜索到的 title\_en匹配到了dynamic模板,就是english分词器,会过滤停用词,is这种停用词就会被过滤掉,用is来搜索就搜索不到了

(3) 定制自己的default mapping template (index level)

```
PUT /my_index { "mappings": { "default": { "all": { "enabled": false } }, "blog": { "all": { "enabled": true } } } }
```

# 索引管理

### 1、重建索引

一个field的设置是不能被修改的,如果要修改一个Field,那么应该重新按照新的mapping,建立一个index,然后将数据批量查询出来,重新用bulk api写入index中

批量查询的时候,建议采用scroll api,并且采用多线程并发的方式来reindex数据,每次scoll就查询 指定日期的一段数据,交给一个线程即可

(1) 一开始,依靠dynamic mapping,插入数据,但是不小心有些数据是2017-01-01这种日期格式的,所以title这种field被自动映射为了date类型,实际上它应该是string类型的

PUT /my\_index/my\_type/3 { "title": "2017-01-03" }

{ "my\_index": { "mappings": { "my\_type": { "properties": { "title": { "type": "date" } } } } }

### (2) 当后期向索引中加入string类型的title值的时候,就会报错

PUT /my\_index/my\_type/4 { "title": "my first article" }

{ "error": { "root\_cause": [ { "type": "mapper\_parsing\_exception", "reason": "failed to parse [title]" } ], "type": "mapper\_parsing\_exception", "reason": "failed to parse [title]", "caused\_by": { "type": "illegal\_argument\_exception", "reason": "Invalid format: "my first article"" } }, "status": 400 }

#### (3) 如果此时想修改title的类型,是不可能的

PUT /my\_index/\_mapping/my\_type { "properties": { "title": { "type": "text" } } }

{ "error": { "root\_cause": [ { "type": "illegal\_argument\_exception", "reason": "mapper [title] of different type, current\_type [date], merged\_type [text]" } ], "type": "illegal\_argument\_exception", "reason": "mapper [title] of different type, current\_type [date], merged\_type [text]" }, "status": 400 }

- (4) 此时,唯一的办法,就是进行reindex,也就是说,重新建立一个索引,将旧索引的数据查询出来,再导入新索引
- (5) 如果说旧索引的名字,是old\_index,新索引的名字是new\_index,终端java应用,已经在使用old\_index在操作了,难道还要去停止java应用,修改使用的index为new\_index,才重新启动java应用吗?这个过程中,就会导致java应用停机,可用性降低
- (6)所以说,给java应用一个别名,这个别名是指向旧索引的,java应用先用着,java应用先用goods\_index alias来操作,此时实际指向的是旧的my\_index

PUT /my\_index/\_alias/goods\_index

(7) 新建一个index, 调整其title的类型为string

PUT /my\_index\_new { "mappings": { "my\_type": { "properties": { "title": { "type": "text" } } } }

(8) 使用scroll api将数据批量查询出来

GET /my\_index/search?scroll=1m { "query": { "match\_all": {} }, "sort": ["doc"], "size": 1 }

#### { "scroll\_id":

"DnF1ZXJ5VGhlbkZldGNoBQAAAAAADpAFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAAAAAAQRY0b25
zVFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAAAAOklWNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAAAADpDFjRvbn
NUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAAAAABY0b25zVFlWWlRqR3ZJajlfc3BXejJ3", "took": 1,
"timed\_out": false, "shards": { "total": 5, "successful": 5, "failed": 0 }, "hits": { "total": 3,
"max\_score": null, "hits": [ { "index": "my\_index", "type": "my\_type", "id": "2", "score": null,

"\_source": { "title": "2017-01-02" }, "sort": [ 0 ] } ] } }

(9) 采用bulk api将scoll查出来的一批数据,批量写入新索引

POST /bulk { "index": { "index": "my\_index\_new", "type": "my\_type", "id": "2" }} { "title": "2017-01-02" }

- (10) 反复循环8~9, 查询一批又一批的数据出来, 采取bulk api将每一批数据批量写入新索引
- (11)将goods\_index alias切换到my\_index\_new上去,java应用会直接通过index别名使用新的索引中的数据,java应用程序不需要停机,零提交,高可用

POST /\_aliases { "actions": [ { "remove": { "index": "my\_index", "alias": "goods\_index" }}, { "add": { "index": "my\_index\_new", "alias": "goods\_index" }} ] }

(12) 直接通过goods\_index别名来查询,是否ok

GET /goods\_index/my\_type/\_search

### 2、基于alias对client透明切换index

PUT /my\_index\_v1/\_alias/my\_index

client对my\_index进行操作

reindex操作,完成之后,切换v1到v2

POST /\_aliases { "actions": [ { "remove": { "index": "my\_index\_v1", "alias": "my\_index" }}, { "add": { "index": "my\_index\_v2", "alias": "my\_index" }} ] }

# 倒排索引

倒排索引,是适合用于进行搜索的

倒排索引的结构

- (1) 包含这个关键词的document list (2) 包含这个关键词的所有document的数量: IDF (inverse document frequency) (3) 这个关键词在每个document中出现的次数: TF (term frequency)
- (4) 这个关键词在这个document中的次序(5)每个document的长度:length norm(6)包含这个关键词的所有document的平均长度

word doc1 doc2

dog \* \* hello \* you \*

倒排索引不可变的好处

(1) 不需要锁,提升并发能力,避免锁的问题(2)数据不变,一直保存在os cache中,只要cache 内存足够(3)filter cache一直驻留在内存,因为数据不变(4)可以压缩,节省cpu和io开销

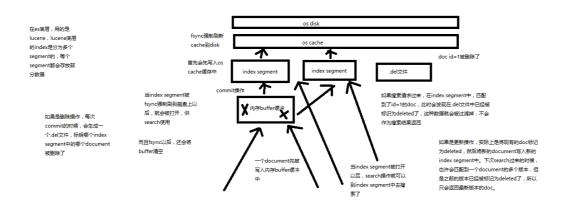
倒排索引不可变的坏处:每次都要重新构建整个索引

### document 写入过程

#### 课程大纲

(1) 数据写入buffer(2)commit point(3)buffer中的数据写入新的index segment(4)等待在os cache中的index segment被fsync强制刷到磁盘上(5)新的index sgement被打开,供search使用(6)buffer被清空

每次commit point时,会有一个.del文件,标记了哪些segment中的哪些document被标记为deleted 了 搜索的时候,会依次查询所有的segment,从旧的到新的,比如被修改过的document,在旧的 segment中,会标记为deleted,在新的segment中会有其新的数据



现有流程的问题,每次都必须等待fsync将segment刷入磁盘,才能将segment打开供search使用,这样的话,从一个document写入,到它可以被搜索,可能会超过1分钟!!!这就不是近实时的搜索了!!!主要瓶颈在于fsync实际发生磁盘IO写数据进磁盘,是很耗时的。

#### 写入流程别改进如下:

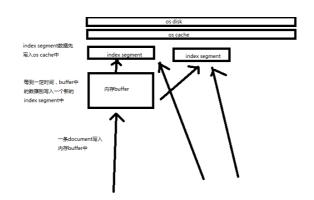
(1) 数据写入buffer (2) 每隔一定时间,buffer中的数据被写入segment文件,但是先写入os cache (3) 只要segment写入os cache,那就直接打开供search使用,不立即执行commit

数据写入os cache,并被打开供搜索的过程,叫做refresh,默认是每隔1秒refresh一次。也就是说,每隔一秒就会将buffer中的数据写入一个新的index segment file,先写入os cache中。所以,es是近实时的,数据写入到可以被搜索,默认是1秒。

POST /my\_index/\_refresh,可以手动refresh,一般不需要手动执行,没必要,让es自己搞就可以了比如说,我们现在的时效性要求,比较低,只要求一条数据写入es,一分钟以后才让我们搜索到就可以了,那么就可以调整refresh interval

PUT /my\_index { "settings": { "refresh\_interval": "30s" } }

这个时候,改进流程的地方到了,我 们不会等待fsync将os cache中的数 据解的os disk,才将index segment 打开供search使用。而是index segment数据一到os cache中,就立 即打开,供search使用



每个秒,buffer被剔 新到一个新的index segment中,所以每 秒都会产生一个新的 index segment file

#### 再次优化的写入流程

(1) 数据写入buffer缓冲和translog日志文件(2)每隔一秒钟,buffer中的数据被写入新的 segment file,并进入os cache,此时segment被打开并供search使用(3)buffer被清空(4)重复 1~3,新的segment不断添加,buffer不断被清空,而translog中的数据不断累加(5)当translog长度达到一定程度的时候,commit操作发生(5-1)buffer中的所有数据写入一个新的segment,并写入os cache,打开供使用(5-2)buffer被清空(5-3)一个commit ponit被写入磁盘,标明了所有的index segment(5-4)filesystem cache中的所有index segment file缓存数据,被fsync强行刷到磁盘上(5-5)现有的translog被清空,创建一个新的translog

基于translog和commit point,如何进行数据恢复

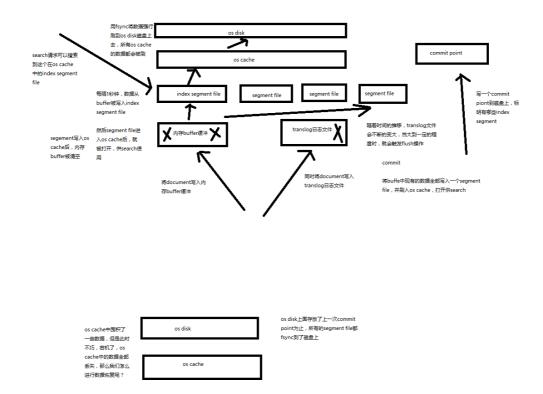
fsync+清空translog,就是flush,默认每隔30分钟flush一次,或者当translog过大的时候,也会flush

POST /my\_index/\_flush, 一般来说别手动flush, 让它自动执行就可以了

translog,每隔5秒被fsync一次到磁盘上。在一次增删改操作之后,当fsync在primary shard和 replica shard都成功之后,那次增删改操作才会成功

但是这种在一次增删改时强行fsync translog可能会导致部分操作比较耗时,也可以允许部分数据丢失,设置异步fsync translog

PUT /my\_index/\_settings { "index.translog.durability": "async", "index.translog.sync\_interval": "5s" }



translog就存储了上一次

的数据的变更记录

flush (commit point) 直到现在最近

每秒一个segment file,文件过多,而且每次search都要搜索所有的segment,很耗时

机器被重启,disk上的数据并没有丢失,此时就会将

translog文件中的变更记录进行回放,重新执行之前的各种

操作,在buffer中执行,再重新剧一个一个的segment到os

默认会在后台执行segment merge操作,在merge的时候,被标记为deleted的document也会被彻底物理删除

#### 每次merge操作的执行流程

(1) 选择一些有相似大小的segment, merge成一个大的segment (2) 将新的segment flush到磁盘上去(3)写一个新的commit point,包括了新的segment,并且排除旧的那些segment (4) 将新的segment打开供搜索(5)将旧的segment删除

