主讲老师：Fox

> 有道笔记：https://note.youdao.com/s/A9NcBG5g

ES要掌握什么：

1. 使用：搜索和聚合操作语法，理解分词，倒排索引，相关性算分（文档匹配度）
2. 优化： 数据预处理，文档建模，集群架构优化，读写性能优化

1. ES数据预处理

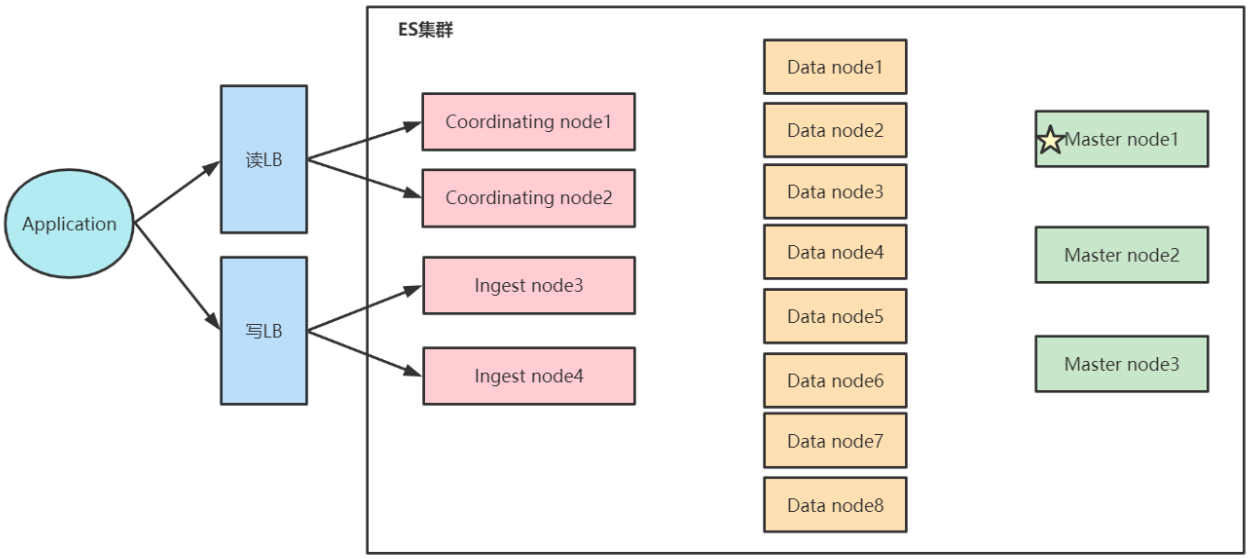
1.1 Ingest Node

Elasticsearch 5.0后，引入的一种新的节点类型。默认配置下，每个节点都是Ingest Node：

- 具有预处理数据的能力，可拦截Index或 Bulk API的请求
- 对数据进行转换，并重新返回给Index或 Bulk API

无需Logstash，就可以进行数据的预处理，例如：

- 为某个字段设置默认值;
- 重命名某个字段的字段名;
- 对字段值进行Split 操作
- 支持设置Painless脚本，对数据进行更加复杂的加工



Ingest Node VS Logstash

	Logstash	Ingest Node
数据输入与输出	支持从不同的数据源读取，并写入不同的数据源	支持从ES REST API获取数据，并且写入Elasticsearch

数据缓冲	实现了简单的数据队列，支持重写	不支持缓冲
数据处理	支持大量的插件，也支持定制开发	内置的插件，可以开发Plugin进行扩展(Plugin更新需要重启)
配置和使用	增加了一定的架构复杂度	无需额外部署

1.2 Ingest Pipeline

应用场景： 修复与增强写入数据

案例

需求：后期需要对Tags进行Aggregation统计。Tags字段中，逗号分隔的文本应该是数组，而不是一个字符串。

```
#Blog数据，包含3个字段，tags用逗号间隔
PUT tech_blogs/_doc/1
{
  "title":"Introducing big data......",
  "tags":"hadoop,elasticsearch,spark",
  "content":"You konw, for big data"
}
```

Pipeline & Processor

- Pipeline ——管道会对通过的数据(文档)，按照顺序进行加工
- Processor——Elasticsearch 对一些加工的行为进行了抽象包装。

Elasticsearch 有很多内置的Processors，也支持通过插件的方式，实现自己的Processor。

https://www.elastic.co/guide/en/elasticsearch/reference/7.17/ingest-processors.html

一些内置的Processors：

- Split Processor： 将给定字段值分成一个数组
- Remove / Rename Processor： 移除一个重命名字段
- Append： 为商品增加一个新的标签
- Convert：将商品价格，从字符串转换成float 类型
- Date / JSON：日期格式转换，字符串转JSON对象
- Date Index Name Processor：将通过该处理器的文档,分配到指定时间格式的索引中
- Fail Processor：一旦出现异常，该Pipeline 指定的错误信息能返回给用户

- Foreach Process：数组字段，数组的每个元素都会使用到一个相同的处理器
- Grok Processor：日志的日期格式切割
- Gsub / Join / Split：字符串替换 | 数组转字符串/字符串转数组
- Lowercase / upcase：大小写转换

```
# 测试split tags
POST _ingest/pipeline/_simulate
{
  "pipeline": {
    "description": "to split blog tags",
    "processors": [
      {
        "split": {
          "field": "tags",
          "separator": ","
        }
      }
    ]
  },
  "docs": [
    {
      "_index": "index",
      "_id": "1",
      "_source": {
        "title": "Introducing big data......",
        "tags": "hadoop,elasticsearch,spark",
        "content": "You konw, for big data"
      }
    },
    {
      "_index": "index",
      "_id": "2",
      "_source": {
        "title": "Introducing cloud computering",
        "tags": "openstack,k8s",
        "content": "You konw, for cloud"
      }
```

```
    }
  ]
}

#同时为文档，增加一个字段。blog查看量
POST _ingest/pipeline/_simulate
{
  "pipeline": {
    "description": "to split blog tags",
    "processors": [
      {
        "split": {
          "field": "tags",
          "separator": ","
        }
      },
      {
        "set":{
          "field": "views",
          "value": 0
        }
      }
    ]
  },

  "docs": [
    {
      "_index":"index",
      "_id":"1",
      "_source":{
        "title":"Introducing big data......",
        "tags":"hadoop,elasticsearch,spark",
        "content":"You konw, for big data"
      }
    },
    {
      "_index":"index",
      "_id":"2",
      "_source":{
        "title":"Introducing cloud computering",
```

```
        "tags":"openstack,k8s",
        "content":"You konw, for cloud"
      }
    }

    ]
}
```

创建pipeline

```
# 为ES添加一个 Pipeline
PUT _ingest/pipeline/blog_pipeline
{
  "description": "a blog pipeline",
  "processors": [
      {
        "split": {
          "field": "tags",
          "separator": ","
        }
      },

      {
        "set":{
          "field": "views",
          "value": 0
        }
      }
    ]
}

#查看Pipleline
GET _ingest/pipeline/blog_pipeline

```

使用pipeline更新数据

```
#不使用pipeline更新数据
PUT tech_blogs/_doc/1
{
  "title":"Introducing big data......",
  "tags":"hadoop,elasticsearch,spark",
  "content":"You konw, for big data"
}

#使用pipeline更新数据
PUT tech_blogs/_doc/2?pipeline=blog_pipeline
{
  "title": "Introducing cloud computering",
  "tags": "openstack,k8s",
  "content": "You konw, for cloud"
}
```

借助update_by_query更新已存在的文档

```
#update_by_query 会导致错误
POST tech_blogs/_update_by_query?pipeline=blog_pipeline
{
}

#增加update_by_query的条件
POST tech_blogs/_update_by_query?pipeline=blog_pipeline
{
    "query": {
        "bool": {
            "must_not": {
                "exists": {
                    "field": "views"
                }
            }
        }
    }
}
```

```
19
20 GET tech_blogs/_search
```

1.3 Painless Script

自Elasticsearch 5.x后引入，专门为Elasticsearch 设计，扩展了Java的语法。Painless支持所有Java的数据类型及Java API子集。

Painless Script具备以下特性：

- 高性能/安全
- 支持显示类型或者动态定义类型

Painless的用途：

- 可以对文档字段进行加工处理
  - 更新或删除字段，处理数据聚合操作
  - Script Field:对返回的字段提前进行计算
  - Function Score:对文档的算分进行处理
- 在Ingest Pipeline中执行脚本
- 在Reindex API，Update By Query时，对数据进行处理

通过Painless脚本访问字段

上下文	语法
Ingestion	ctx.field_name
Update	ctx._source.field_name
Search & Aggregation	doc["field_name"]

测试

```
1 # 增加一个 Script Prcessor
2 POST _ingest/pipeline/_simulate
3 {
4   "pipeline": {
```

```
    "description": "to split blog tags",
    "processors": [
      {
        "split": {
          "field": "tags",
          "separator": ","
        }
      },
      {
        "script": {
          "source": """
          if(ctx.containsKey("content")){
            ctx.content_length = ctx.content.length();
          }else{
            ctx.content_length=0;
          }

          """
        }
      },

      {
        "set":{
          "field": "views",
          "value": 0
        }
      }
    ]
  },

  "docs": [
    {
      "_index":"index",
      "_id":"1",
      "_source":{
        "title":"Introducing big data......",
  "tags":"hadoop,elasticsearch,spark",
  "content":"You konw, for big data"
      }
    },
```

```
45
46
47      {
48        "_index":"index",
49        "_id":"2",
50        "_source":{
51          "title":"Introducing cloud computering",
52    "tags":"openstack,k8s",
53    "content":"You konw, for cloud"
54        }
55      }
56
57      ]
58  }
59
60  DELETE tech_blogs
61  PUT tech_blogs/_doc/1
62  {
63    "title":"Introducing big data......",
64    "tags":"hadoop,elasticsearch,spark",
65    "content":"You konw, for big data",
66    "views":0
67  }
68
69  POST tech_blogs/_update/1
70  {
71    "script": {
72      "source": "ctx._source.views += params.new_views",
73      "params": {
74        "new_views":100
75      }
76    }
77  }
78
79  # 查看views计数
80  POST tech_blogs/_search
81
82
83
```

```
#保存脚本在 Cluster State
POST _scripts/update_views
{
  "script":{
    "lang": "painless",
    "source": "ctx._source.views += params.new_views"
  }
}

POST tech_blogs/_update/1
{
  "script": {
    "id": "update_views",
    "params": {
      "new_views":1000
    }
  }
}


GET tech_blogs/_search
{
  "script_fields": {
    "rnd_views": {
      "script": {
        "lang": "painless",
        "source": """
          java.util.Random rnd = new Random();
          doc['views'].value+rnd.nextInt(1000);
        """
      }
    }
  },
  "query": {
    "match_all": {}
  }
}
```

2. ES文档建模

2.1 Elasticsearch中如何处理关联关系

关系型数据库范式化（Normalize）设计的主要目标是减少不必要的更新，往往会带来一些副作用：

- 一个完全范式化设计的数据库会经常面临“查询缓慢”的问题。数据库越范式化，就需要Join越多的表；
- 范式化节省了存储空间，但是存储空间已经变得越来越便宜；
- 范式化简化了更新，但是数据读取操作可能更多。

反范式化(Denormalize)的设计不使用关联关系，而是在文档中保存冗余的数据拷贝。

- 优点: 无需处理Join操作，数据读取性能好。Elasticsearch可以通过压缩_source字段，减少磁盘空间的开销
- 缺点: 不适合在数据频繁修改的场景。 一条数据的改动，可能会引起很多数据的更新

关系型数据库，一般会考虑Normalize 数据；在Elasticsearch，往往考虑Denormalize 数据。
Elasticsearch并不擅长处理关联关系，一般会采用以下四种方法处理关联：

- 对象类型
- 嵌套对象(Nested Object)
- 父子关联关系(Parent / Child)
- 应用端关联

2.2 对象类型

案例1： 博客作者信息变更

对象类型：

- 在每一博客的文档中都保留作者的信息
- 如果作者信息发生变化，需要修改相关的博客文档

```
DELETE blog
# 设置blog的 Mapping
PUT /blog
{
  "mappings": {
    "properties": {
      "content": {
```

```
        "type": "text"
      },
      "time": {
        "type": "date"
      },
      "user": {
        "properties": {
          "city": {
            "type": "text"
          },
          "userid": {
            "type": "long"
          },
          "username": {
            "type": "keyword"
          }
        }
      }
    }
  }
}

# 插入一条 blog信息
PUT /blog/_doc/1
{
  "content":"I like Elasticsearch",
  "time":"2022-01-01T00:00:00",
  "user":{
    "userid":1,
    "username":"Fox",
    "city":"Changsha"
  }
}


# 查询 blog信息
POST /blog/_search
{
  "query": {
    "bool": {
```

```
      "must": [
        {"match": {"content": "Elasticsearch"}},
        {"match": {"user.username": "Fox"}}
      ]
    }
  }
}
```

案例2：包含对象数组的文档

```
DELETE /my_movies

# 电影的Mapping信息
PUT /my_movies
{
      "mappings" : {
      "properties" : {
        "actors" : {
          "properties" : {
            "first_name" : {
              "type" : "keyword"
            },
            "last_name" : {
              "type" : "keyword"
            }
          }
        },
        "title" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        }
```

```
      }
    }
}


# 写入一条电影信息
POST /my_movies/_doc/1
{
  "title":"Speed",
  "actors":[
    {
      "first_name":"Keanu",
      "last_name":"Reeves"
    },

    {
      "first_name":"Dennis",
      "last_name":"Hopper"
    }

  ]
}

# 查询电影信息
POST /my_movies/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"actors.first_name": "Keanu"}},
        {"match": {"actors.last_name": "Hopper"}}
      ]
    }
  }

}
```

思考：为什么会搜到不需要的结果？

存储时，内部对象的边界并没有考虑在内,JSON格式被处理成扁平式键值对的结构。当对多个字段进行查询时，导致了意外的搜索结果。可以用Nested Data Type解决这个问题。

```
"title":"Speed"
"actor".first_name: ["Keanu","Dennis"]
"actor".last_name: ["Reeves","Hopper"]
```

2.3 嵌套对象(Nested Object)

什么是Nested Data Type

- Nested数据类型: 允许对象数组中的对象被独立索引
- 使用nested 和properties 关键字，将所有actors索引到多个分隔的文档
- 在内部, Nested文档会被保存在两个Lucene文档中，在查询时做Join处理

```
DELETE /my_movies
# 创建 Nested 对象 Mapping
PUT /my_movies
{
      "mappings" : {
      "properties" : {
        "actors" : {
          "type": "nested",
          "properties" : {
            "first_name" : {"type" : "keyword"},
            "last_name" : {"type" : "keyword"}
          }},
        "title" : {
          "type" : "text",
          "fields" : {"keyword":{"type":"keyword","ignore_above":256}}
        }
      }
    }
}

```

```
POST /my_movies/_doc/1
{
  "title":"Speed",
  "actors":[
    {
      "first_name":"Keanu",
      "last_name":"Reeves"
    },

    {
      "first_name":"Dennis",
      "last_name":"Hopper"
    }

  ]
}

# Nested 查询
POST /my_movies/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {"title": "Speed"}},
        {
          "nested": {
            "path": "actors",
            "query": {
              "bool": {
                "must": [
                  {"match": {
                    "actors.first_name": "Keanu"
                  }},

                  {"match": {
                    "actors.last_name": "Hopper"
                  }}
                ]
              }
            }
```

```
          }
        }
      ]
    }
  }
}

# Nested Aggregation
POST /my_movies/_search
{
  "size": 0,
  "aggs": {
    "actors_agg": {
      "nested": {
        "path": "actors"
      },
      "aggs": {
        "actor_name": {
          "terms": {
            "field": "actors.first_name",
            "size": 10
          }
        }
      }
    }
  }
}


# 普通 aggregation不工作
POST /my_movies/_search
{
  "size": 0,
  "aggs": {
    "actors_agg": {
      "terms": {
        "field": "actors.first_name",
        "size": 10
      }
    }
```

```
  }
}
```

2.4 父子关联关系(Parent / Child)

- 对象和Nested对象的局限性: 每次更新，可能需要重新索引整个对象(包括根对象和嵌套对象)
- ES提供了类似关系型数据库中Join 的实现。使用Join数据类型实现，可以通过维护Parent/ Child的关系，从而分离两个对象
  - 父文档和子文档是两个独立的文档
  - 更新父文档无需重新索引子文档。子文档被添加，更新或者删除也不会影响到父文档和其他的子文档

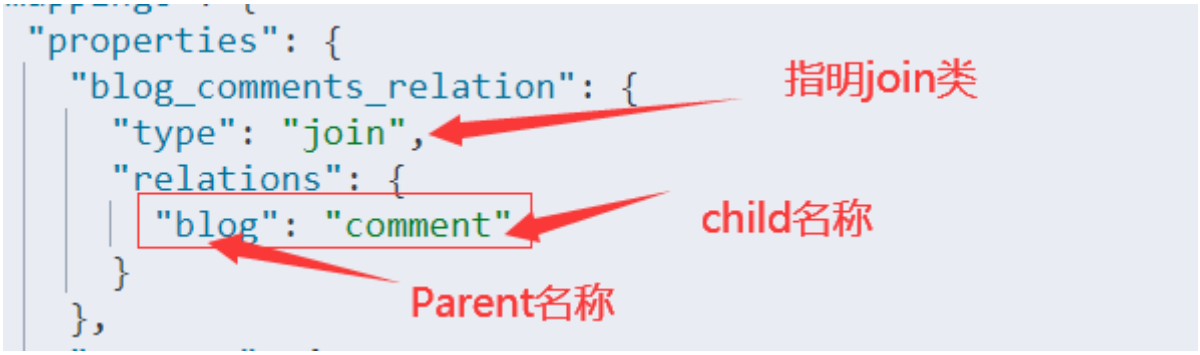
设定 Parent/Child Mapping

```
DELETE /my_blogs

# 设定 Parent/Child Mapping
PUT /my_blogs
{
  "settings": {
    "number_of_shards": 2
  },
  "mappings": {
    "properties": {
      "blog_comments_relation": {
        "type": "join",
        "relations": {
          "blog": "comment"
        }
      },
      "content": {
        "type": "text"
      },
      "title": {
        "type": "keyword"
      }
    }
```

```
  }
}

```



索引父文档

```
#索引父文档
PUT /my_blogs/_doc/blog1
{
  "title":"Learning Elasticsearch",
  "content":"learning ELK ",
  "blog_comments_relation":{
    "name":"blog"
  }
}

#索引父文档
PUT /my_blogs/_doc/blog2
{
  "title":"Learning Hadoop",
  "content":"learning Hadoop",
  "blog_comments_relation":{
    "name":"blog"
  }
}
```

索引子文档

```
#索引子文档
PUT /my_blogs/_doc/comment1?routing=blog1
{
  "comment":"I am learning ELK",
  "username":"Jack",
  "blog_comments_relation":{
    "name":"comment",
    "parent":"blog1"
  }
}

#索引子文档
PUT /my_blogs/_doc/comment2?routing=blog2
{
  "comment":"I like Hadoop!!!!!",
  "username":"Jack",
  "blog_comments_relation":{
    "name":"comment",
    "parent":"blog2"
  }
}

#索引子文档
PUT /my_blogs/_doc/comment3?routing=blog2
{
  "comment":"Hello Hadoop",
  "username":"Bob",
```

```
  "blog_comments_relation":{
    "name":"comment",
    "parent":"blog2"
  }
}
```

注意：

- 父文档和子文档必须存在相同的分片上，能够确保查询join 的性能
- 当指定子文档时候，必须指定它的父文档Id。使用routing参数来保证，分配到相同的分片

查询

```
# 查询所有文档
POST /my_blogs/_search

#根据父文档ID查看
GET /my_blogs/_doc/blog2

# Parent Id 查询
POST /my_blogs/_search
{
  "query": {
    "parent_id": {
      "type": "comment",
      "id": "blog2"
    }
  }
}

# Has Child 查询,返回父文档
```

```
POST /my_blogs/_search
{
  "query": {
    "has_child": {
      "type": "comment",
      "query" : {
                "match": {
                    "username" : "Jack"
                }
            }
    }
  }
}


# Has Parent 查询，返回相关的子文档
POST /my_blogs/_search
{
  "query": {
    "has_parent": {
      "parent_type": "blog",
      "query" : {
                "match": {
                    "title" : "Learning Hadoop"
                }
            }
    }
  }
}

#通过ID ， 访问子文档
GET /my_blogs/_doc/comment3
#通过ID和routing ， 访问子文档
GET /my_blogs/_doc/comment3?routing=blog2

#更新子文档
PUT /my_blogs/_doc/comment3?routing=blog2
{
    "comment": "Hello Hadoop??",
    "blog_comments_relation": {
```

```
        "name": "comment",
        "parent": "blog2"
    }
}
```

嵌套文档 VS 父子文档

	Nested Object	Parent / Child
优点	文档存储在一起，读取性能高	父子文档可以独立更新
缺点	更新嵌套的子文档时，需要更新整个文档	需要额外的内存维护关系。读取性能相对差
适用场景	子文档偶尔更新，以查询为主	子文档更新频繁

2.5 ElasticSearch数据建模最佳实践

如何处理关联关系

- Object: 优先考虑反范式（Denormalization）
- Nested: 当数据包含多数值对象，同时有查询需求
- Child/Parent：关联文档更新非常频繁时

避免过多字段

- 一个文档中，最好避免大量的字段
  - 过多的字段数不容易维护
  - Mapping 信息保存在Cluster State 中，数据量过大，对集群性能会有影响
  - 删除或者修改数据需要reindex
- 默认最大字段数是1000，可以设置index.mapping.total_fields.limit限定最大字段数。·

思考：什么原因会导致文档中有成百上千的字段?

生产环境中，尽量不要打开 Dynamic，可以使用Strict控制新增字段的加入

- true ： 未知字段会被自动加入

- false： 新字段不会被索引，但是会保存在_source
- strict： 新增字段不会被索引，文档写入失败

对于多属性的字段，比如cookie，商品属性，可以考虑使用Nested

避免正则，通配符，前缀查询

正则，通配符查询，前缀查询属于Term查询，但是性能不够好。特别是将通配符放在开头，会导致性能的灾难

案例：针对版本号的搜索

```
# 将字符串转对象
PUT softwares/
{
  "mappings": {
    "properties": {
      "version": {
        "properties": {
          "display_name": {
            "type": "keyword"
          },
          "hot_fix": {
            "type": "byte"
          },
          "marjor": {
            "type": "byte"
          },
          "minor": {
            "type": "byte"
          }
        }
      }
    }
  }
}


#通过 Inner Object 写入多个文档
PUT softwares/_doc/1
```

```
{
  "version":{
  "display_name":"7.1.0",
  "marjor":7,
  "minor":1,
  "hot_fix":0
  }

}

PUT softwares/_doc/2
{
  "version":{
  "display_name":"7.2.0",
  "marjor":7,
  "minor":2,
  "hot_fix":0
  }
}

PUT softwares/_doc/3
{
  "version":{
  "display_name":"7.2.1",
  "marjor":7,
  "minor":2,
  "hot_fix":1
  }
}


# 通过 bool 查询,
POST softwares/_search
{
  "query": {
    "bool": {
      "filter": [
        {
          "match":{
            "version.marjor":7
```

```
          }
        },
        {
          "match":{
            "version.minor":2
          }
        }
      ]
    }
  }
}

```

避免空值引起的聚合不准

```
# Not Null 解决聚合的问题
DELETE /scores
PUT /scores
{
  "mappings": {
      "properties": {
        "score": {
          "type": "float",
          "null_value": 0
        }
      }
    }
}

PUT /scores/_doc/1
{
 "score": 100
}
PUT /scores/_doc/2
{
 "score": null
```

```
}

POST /scores/_search
{
  "size": 0,
  "aggs": {
    "avg": {
      "avg": {
        "field": "score"
      }
    }
  }
}
```

为索引的Mapping加入Meta 信息

- Mappings设置非常重要，需要从两个维度进行考虑
  - 功能：搜索，聚合，排序
  - 性能：存储的开销; 内存的开销; 搜索的性能
- Mappings设置是一个迭代的过程
  - 加入新的字段很容易（必要时需要update_by_query)
  - 更新删除字段不允许(需要Reindex重建数据)
  - 最好能对Mappings 加入Meta 信息，更好的进行版本管理
  - 可以考虑将Mapping文件上传git进行管理

```
PUT /my_index
{
  "mappings": {
    "_meta": {
      "index_version_mapping": "1.1"
    }
  }
}
```

3. ES读写性能调优

3.1 ES底层读写工作原理分析

写请求是写入 primary shard，然后同步给所有的 replica shard；读请求可以从 primary shard 或 replica shard 读取，采用的是随机轮询算法。

ES写入数据的过程

1. 客户端选择一个node发送请求过去，这个node就是coordinating node (协调节点)
2. coordinating node，对document进行路由，将请求转发给对应的node
3. node上的primary shard处理请求，然后将数据同步到replica node
4. coordinating node如果发现primary node和所有的replica node都搞定之后，就会返回请求到客户端

ES读取数据的过程

根据id查询数据的过程

根据 doc id 进行 hash，判断出来当时把 doc id 分配到了哪个 shard 上面去，从那个 shard 去查询。

1. 客户端发送请求到任意一个 node，成为 coordinate node 。
2. coordinate node 对 doc id 进行哈希路由，将请求转发到对应的 node，此时会使用 round-robin 随机轮询算法，在 primary shard 以及其所有 replica 中随机选择一个，让读请求负载均衡。
3. 接收请求的 node 返回 document 给 coordinate node 。
4. coordinate node 返回 document 给客户端。

根据关键词查询数据的过程

- 客户端发送请求到一个 coordinate node 。
- 协调节点将搜索请求转发到所有的 shard 对应的 primary shard 或 replica shard ，都可以。
- query phase：每个 shard 将自己的搜索结果返回给协调节点，由协调节点进行数据的合并、排序、分页等操作，产出最终结果。
- fetch phase：接着由协调节点根据 doc id 去各个节点上拉取实际的 document 数据，最终返回给客户端。

写数据底层原理

核心概念

segment file: 存储倒排索引的文件，每个segment本质上就是一个倒排索引，每秒都会生成一个segment文件，当文件过多时es会自动进行segment merge（合并文件），合并时会同时将已经标注删除的文档物理删除。
commit point: 记录当前所有可用的segment，每个commit point都会维护一个.del文件，即每个.del文件都有一个commit point文件（es删除数据本质是不属于物理删除），当es做删改操作时首先会在.del文件中声明某个document已经被删除，文件内记录了在某个segment内某个文档已经被删除，当查询请求过来时在segment中被删除的文件是能够查出来的，但是当返回结果时会根据commit point维护的那个.del文件把已经删除的文档过滤掉
translog日志文件: 为了防止elasticsearch宕机造成数据丢失保证可靠存储，es会将每次写入数据同时写到translog日志中。
os cache：操作系统里面，磁盘文件其实都有一个东西，叫做os cache，操作系统缓存，就是说数据写入磁盘文件之前，会先进入os cache，先进入操作系统级别的一个内存缓存中去

Refresh

- 将文档先保存在Index buffer中，以refresh_interval为间隔时间，定期清空buffer，生成 segment,借助文件系统缓存的特性，先将segment放在文件系统缓存中，并开放查询，以提升搜索的实时性

Translog

- Segment没有写入磁盘，即便发生了当机，重启后，数据也能恢复，从ES6.0开始默认配置是每次请求都会落盘

Flush

- 删除旧的translog 文件
- 生成Segment并写入磁盘 | 更新commit point并写入磁盘。ES自动完成，可优化点不多

3.2 如何提升集群的读写性能

提升集群读取性能的方法

数据建模

- 尽量将数据先行计算，然后保存到Elasticsearch 中。尽量避免查询时的 Script计算

```
#避免查询时脚本
GET blogs/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": {
          "title": "elasticsearch"
```

```
        }}
      ],

      "filter": {
        "script": {
          "script": {
            "source": "doc['title.keyword'].value.length()>5"
          }
        }
      }
    }
  }
}
```

- 尽量使用Filter Context，利用缓存机制，减少不必要的算分
- 结合profile，explain API分析慢查询的问题，持续优化数据模型
- 避免使用*开头的通配符查询

```
GET /es_db/_search
{
  "query": {
    "wildcard": {
      "address": {
        "value": "*白云*"
      }
    }
  }
}
```

优化分片

- 避免Over Sharing
  - 一个查询需要访问每一个分片，分片过多，会导致不必要的查询开销
- 结合应用场景，控制单个分片的大小
  - Search: 20GB
  - Logging: 40GB

- Force-merge Read-only索引
  - 使用基于时间序列的索引，将只读的索引进行force merge，减少segment数量

```
#手动force merge
POST /my_index/_forcemerge
```

提升写入性能的方法

- 写性能优化的目标: 增大写吞吐量，越高越好
- 客户端: 多线程，批量写
  - 可以通过性能测试，确定最佳文档数量
  - 多线程: 需要观察是否有HTTP 429（Too Many Requests）返回，实现 Retry以及线程数量的自动调节
- 服务器端: 单个性能问题，往往是多个因素造成的。需要先分解问题，在单个节点上进行调整并且结合测试，尽可能压榨硬件资源,以达到最高吞吐量
  - 使用更好的硬件。观察CPU / IO Block
  - 线程切换 | 堆栈状况

服务器端优化写入性能的一些手段

- 降低IO操作
  - 使用ES自动生成的文档Id
  - 一些相关的ES 配置，如Refresh Interval
- 降低 CPU 和存储开销
  - 减少不必要分词
  - 避免不需要的doc_values
  - 文档的字段尽量保证相同的顺序，可以提高文档的压缩率
- 尽可能做到写入和分片的均衡负载，实现水平扩展
  - Shard Filtering / Write Load Balancer
- 调整Bulk 线程池和队列

注意：ES 的默认设置，已经综合考虑了数据可靠性，搜索的实时性，写入速度，一般不要盲目修改。一切优化，都要基于高质量的数据建模。

建模时的优化

- 只需要聚合不需要搜索，index设置成false
- 不要对字符串使用默认的dynamic mapping。字段数量过多，会对性能产生比较大的影响
- Index_options控制在创建倒排索引时，哪些内容会被添加到倒排索引中。

如果需要追求极致的写入速度，可以牺牲数据可靠性及搜索实时性以换取性能：

- 牺牲可靠性: 将副本分片设置为0，写入完毕再调整回去
- 牺牲搜索实时性：增加Refresh Interval的时间
- 牺牲可靠性: 修改Translog的配置

降低 Refresh的频率

- 增加refresh_interval 的数值。默认为1s ，如果设置成-1，会禁止自动refresh
  - 避免过于频繁的refresh，而生成过多的segment 文件
  - 但是会降低搜索的实时性

```
PUT /my_index/_settings
{
    "index" : {
        "refresh_interval" : "10s"
    }
}
```

- 增大静态配置参数indices.memory.index_buffer_size
  - 默认是10%，会导致自动触发refresh

降低Translog写磁盘的频率，但是会降低容灾能力

- Index.translog.durability: 默认是request，每个请求都落盘。设置成async，异步写入
- Index.translog.sync_interval：设置为60s，每分钟执行一次
- Index.translog.flush_threshod_size: 默认512 m，可以适当调大。当translog 超过该值，会触发flush

分片设定

- 副本在写入时设为0，完成后再增加
- 合理设置主分片数，确保均匀分配在所有数据节点上

- Index.routing.allocation.total_share_per_node:限定每个索引在每个节点上可分配的主分片数

调整Bulk 线程池和队列

- 客户端
  - 单个bulk请求体的数据量不要太大，官方建议大约5-15m
  - 写入端的 bulk请求超时需要足够长，建议60s 以上
  - 写入端尽量将数据轮询打到不同节点。
- 服务器端
  - 索引创建属于计算密集型任务，应该使用固定大小的线程池来配置。来不及处理的放入队列，线程数应该配置成CPU核心数+1，避免过多的上下文切换
  - 队列大小可以适当增加，不要过大，否则占用的内存会成为GC的负担
  - ES线程池设置： https://blog.csdn.net/justlpf/article/details/103233215

```
DELETE myindex
PUT myindex
{
  "settings": {
    "index": {
      "refresh_interval": "30s",  #30s一次refresh
      "number_of_shards": "2"
    },
    "routing": {
      "allocation": {
        "total_shards_per_node": "3"  #控制分片，避免数据热点
      }
    },
    "translog": {
      "sync_interval": "30s",
      "durability": "async"     #降低translog落盘频率
    },
    "number_of_replicas": 0
  },
  "mappings": {
    "dynamic": false,     #避免不必要的字段索引，必要时可以通过update by query
索引必要的字段
    "properties": {}
```

```
24      }
25  }
```