# 基于 Elasticsearch 集群的数据查询优化

杨樾人，3160104875

浙江大学计算机学院软件工程专业

**摘　要**：本文介绍了在软件工程项目实训过程中 Elasticsearch 集群的使用心得，包括 Elasticsearch 的集群的配置和参数调优，以及与 mogodb 的对比。

**关键词：** Elasticsearch; Search Engine

## Data query optimization based on Elasticsearch cluster

Yang Yueren, 3160104875.

**Abstract:** This article describes the use of Elasticsearch clusters in the software engineering project training process, including Elasticsearch cluster configuration and parameter tuning, and comparison with mogodb.

**Key words:** Elasticsearch; Search Engine

# Table of Contents

# 一、背景介绍

随着互联网的不断发展，网络流行语已经成了年轻人表达情感的主要语言方式之一。它被认同，反映某种现实，类似大碗宽面、这些网络用语，生动形象地体现了网民们当时的心情，也被用来表达情感。基于以上的现状，网络流行语的聚合搜索引擎以小鸡词典、b 站、微博等网络平台的新词热梗为数据来源，以 Elasticsearch 作为主要搜索引擎，为渴望与时俱进使用新梗的广大网友提供搜索平台，在遇到新词时进行方便的搜索。

Elasticsearch 是一个基于 Lucene 的分布式全文搜索引擎，能够横向扩展数以百计的服务器，存储 PB 级的数据，而且对每个字段都可以建立索引并且检索，并且可以在极短的时间内存储、搜索和分析大量的数据，程序员最爱的网站 Github 的搜索就是基于 ES 构建的，GitHub 大约有 30TB 的索引文件数据，由此可见 Elasticsearch（下文简称 ES）强大的搜索功能。

在此次的深度搜索引擎项目之中，虽然 Elasticsearch 也可以在一个节点上使用，该节点可以同时担任 master node 和 data node，但是为了发挥 Elasticsearch 的分布式搜索的优势，在我们的深度搜索引擎中我们使用了三台服务器提供 Elasticsearch 的服务。下文将详细介绍从集群部署到优化查询的一些要点。

# 二、Elasticsearch 的集群部署

## 1. ES 概念简介

Elasticsearch 中有几个比较重要的概念，集群是指连接在一起的若干台服务器，不同的服务器承担不同的角色，一起提供服务。集群中有主节点、数据节点和客户端节点等。主节点负责管理整个集群，当群集的拓扑结构改变时把索引分片分派到相应的节点上，主节点是从可以担任主节点的节点中选举出来的。数据节点只负责存储数据，客户端节点在选举主节点过程中起作用。ES 的分片是把索引信息分散到多个节点上，相当于一桶水用多个杯子装。副本是指索引信息的拷贝。

在进行 ES 的配置时，首先要考虑节点数和分片数。通过实验，多节点的 ES 集群中的节点数至少为 3，分片数为一倍的节点数量到两倍的节点数量。

节点数和分片数相等时，每个节点负责一个分片的检索，ES 集群的性能可以达到最优。对于一个 3 节点集群，为每个节点分配一个分片，总共 3 个分片。但是由于 ES 的不可变性的限制，系统无法对分片进行重新拆分分配，除非重新索引这个文件集合。但是我在三个节点的集群中再加入一个节点，这时候分片数量小于了节点数，在搜索上效率会降低，所以为了支持水平扩展，可以为集群分配比节点数更多的分片数，也就是说每个节点有多个分片。但是每个节点有多个分片时，需要考虑性能的问题，每个节点最好不要超过两个分片，

我采用了官方给默认配置中分片数目为 5，这样既可以拓展到 5 个节点，也可以保证性能。

我的集群的其余配置如下：

```
#节点集群名称和节点类型
cluster.name: yang-es-clusters
node.name: node-3
node.master: true
node.data: true
#同个集群其他节点的信息，ES 通过广播的方式寻找同一集群的其他节点
discovery.zen.ping.unicast.hosts: [ "0.0.0.0", "106.14.191.xxx",
"120.79.191.xxx"]
#选举主节点时需要由至少 2 个节点参与投票
discovery.zen.minimum_master_nodes: 2
gateway.recover_after_nodes: 1
#配置本节点的 ip，默认开发 9300 端口用于节点间 TCP 通信
network.host: 0.0.0.0
network.publish_host: 106.14.227.30
network.bind_host: 0.0.0.0
```

## 2. Elsaticsearch 集群至少需要有三个节点

上文写到我们组搭建的 ES 集群使用了三台服务器，这也是搭建 ES 集群所需的最少节点数，是因为需要防止 ES 集群发生脑裂。ES 中维护索引状态最重要的节点是主节点，主节点是被投票选举出来的。

### 2.1 三个和尚投票

当主节点出现问题，从节点不能与主节点通信时，从节点会发起选举任命新的主节点，同时新的主节点会接管旧的主节点的所有工作，如果旧的主节点重新恢复并加入到集群中，新的主节点会将原来旧的主节点降级为从节点，这样就不会有冲突发生。所有这个过程都由 ES 自己处理，使用者无需任何参与。

### 2.2 两个和尚投票

但是，当只有两个节点的时候，一主（master）一从（slave），如果主从直接的通信出现问题时，从节点 slave 会自我提升为 master，但是当恢复通信时，我们就会同时有两个 master。因为此时，对于原来的主节点角度考虑，它认为是原来的从节点出现问题，现在仍然需要作为 slave 重新加入。这样，两个节点的时候，我们就出现了集群不知道将哪个节点选举为主节点的情况，也就是我们通常说的“分脑”。

为了防止这种情况的发生，第三个节点的出现会打破平衡，解决冲突问题。

### 2.3 三个和尚仍然存在问题

分脑的问题同样会出现在具有三或三个以上节点的集群中，为了降低发生的概率，ElasticSearch 提供了一个配置 ***discovery.zen.minimum_master_nodes*** 它规定了在选举新的 master 时，一个集群下最少需要的节点数。例如，一个 3 节点集群，这个数字为 2，2 个节点可以防止单个节点在脱离集群时，将其自己选举成 master，相反，它会等待直到重新加入到集群中。这个数值可以通过一个公式确定：

这里的配置是指当主节点宕掉掉时候至少同时需要几个节点才重新进行投票选举新的主节点，官方建议将此数目配置为`(N / 2 + 1)`，可以有效的防止脑裂。

从上文和尚投票的分析可以得知，如果只有两个节点，当这两个节点通讯故障的时候，会各自选举自己为主节点，而当通讯恢复正常时候会发生冲突，所以只有超过一半的节点投票选举出来的主节点才可以掌控整个集群。

我在配置 ES 的时候，主节点所在的服务器由于网络问题，经常会发生断网的现象，此时集群的状态会由绿色（正常）转变为红色（预警）状态，而当非主节点宕机的时候，集群状态会变为黄色（所有的主分片可用，但是副本分片不可用）。这个问题的解决方案只有一种，设置容易宕机的节点为数据节点，禁止其被选举为主节点。

# 二、Elasticsearch 的查询参数优化

## 1. Lucene 的打分模型

$$score(q,d) = coord(q,d) * queryNorm(q) * \sum_{t\ in\ q} (tf(t\ in\ d) * idf(t)^2 * boost(t) * norm(t,d))$$

图 1 Lecune 的打分模型[1]

由于 ES 是基于 Lucene，所以 ES 也是使用的打分机制。通过上面的公式，一篇文档的分数实际上是由查询语句 q 和文档 d 作为变量的一个函数值。打分公式中有两部分不直接依赖于查询词，它们是 coord 和 queryNorm。公式的值是这样计算的，coord 和 queryNorm 两大部分直接乘以查询语句中每个查询词计算值的总和。另一方面，这个总和也是由每个查询词的词频(tf)，逆文档频率(idf)，查询词的权重，还有 norm，也就是前面说的 length norm 相乘而得的结果。

从中可以得出以下几条规则：

- 匹配到的关键词越稀有，文档的得分就越高。
- 文档的域越小(包含比较少的 Term)，文档的得分就越高。
- 设置的权重(索引和搜索时设置的都可以)越大，文档得分越高。

随着 Lucene 的发展，打分模型也引入了新的相似度模型，并且可以在 ES 中指定，现在比较流行的是 Okapi BM25，Divergence from randomness 和 Information based。

BM25 是基于概率模型的相似度模型，适合处理短文本，关键词的重复次数对整个文档得分影响比较大。DFR 和 IB 比较类似，基于同名概率模型，适用于自然语言类的文本。我们的搜索引擎要搜索的字段比较少，内容也是以短文本为主，并且倾向于能够对名字和标签进行准确匹配，如果关键词在内容中多次重复，明显词条是用户所查询的结果，所以 BM25 更加适合我们的搜索引擎。

## 2. 分词器的选择

ES 是基于词的搜索引擎，其能够快速的通过搜索词检索出对应的文章归功于倒排索引，使用不同的分词器对于检索效果也有重大影响。

ES 的默认分词器对英文句子的切割效果比较好，但用于中文句子的分割时，只会将句子分割成孤立的一个个的字，所以需要指定建立索引时的分词器和搜索分词器。我们使用的是 IKAnalyzer，是目前比较流行的中文分词器之一,设置比较简单,稳定。

在 Sprint Boot 中建立索引时候，对 ES 的支持度不如直接在 ES 里面自己建立索引可操作性高，以下是我建立索引的代码。

```
curl -XPOST http://106.14.227.30:9200/chageng/EntryDb/_mapping -H 'Content-Type:application/json' -d' { "EntryDb": { "properties": {
    "content": {
        "type": "text",
        "analyzer": "ik_max_word",
        "search_analyzer": "ik_smart",
        "fields": {
            "keyword": {
                "type": "text",
                "analyzer": "ik_max_word",
```

```
                "search_analyzer": "ik_smart"
            }
        }
    },
    "imageList": {
        "type": "text",
        "fields": {
            "keyword": {
                "type": "keyword",
                "ignore_above": 256
            }
        }
    },
    "like": {
        "type": "long"
    },
    "name": {
        "type": "text",
        "analyzer": "ik_max_word",
        "search_analyzer": "ik_smart",
        "fields": {
            "keyword": {
                "type": "text",
                "analyzer": "ik_max_word",
                "search_analyzer": "ik_smart"
            }
        }
    }
}'
```

# 3. 查询语句的优化

## 3.1 term、match 与 multi_match

ES 中的 term 是代表完全匹配，也就是精确查询，搜索前不会再对搜索词进行分词，所以我们的搜索词必须是文档分词集合中的一个。以下代码将会在 name 中精确匹配为“小鸡快跑”的词条。

```
curl -XPOST '106.14.227.30: 9200/chageng/_search?pretty' -H 'Content-Type: application/json' -d '
{
  "query":{
    "term":{
        "name":"小鸡快跑"
    }
  }
}'
```

ES 的 match 搜索会先对搜索词进行分词，对于最基本的 match 搜索来说，只要搜索词的分词集合中的一个或多个存在于文档中即可，例如，当我们搜索小鸡快跑，搜索词会先分词为小鸡和快跑,只要文档中包含小鸡和快跑任意一个词，都会被搜索到。

如果文档 1 中有小鸡，文档 2 中有快跑，那么这两个文档都会被检索到，而如果文档 3 中有小鸡和快跑两个词，文档 3 也将被返回，并且文档 3 将被排在首位。所有被返回的文档将依靠_score 的分数进行排序，得分的算法参考上文。

```
curl -XPOST '106.14.227.30: 9200/chageng/_search?pretty' -H 'Content-Type: application/json' -d '
{
    "query": {
        "match": {
            "content": "小鸡快跑"
        }
```

```
    }
}'
```

ES 的 multi_match 是对多个字段进行匹配，其中一个字段包含分词，该文档即可被搜索到并且返回。在实际使用中用的比较多。

```
curl -XPOST '106.14.227.30: 9200/chageng/_search?pretty' -H 'Content-Type: application/json' -d '
{
"query": {
"bool": {
    "must": {
        "match": {
            "tagList": "社交"
        }
    },
    "should": {
        "multi_match": {
            "query": "吓得我瓜子都掉了",
            "type": "best_fields",
            "fields": [
                "name^2",
                "content"
            ],
                "fuzziness": "AUTO",
            "tie_breaker": 0.4,
            "minimum_should_match": "30%"
        }
    }
}
}
}
'
```

### 3.2 组合过滤器 bool

bool 过滤器通过 and 、 or 和 not 逻辑组合将多个过滤器进行组合。bool 查询可以接受 must 、 must_not 和 should 参数下的多个查询语句，对查询结果进行筛选，分别对应 AND NOT OR。bool 查询会为每个文档计算相关度评分 _score ， 再将所有匹配的 must 和 should 语句的分数 _score 求和，最后除以 must 和 should 语句的总数。

must_not 语句不会影响评分； 它的作用只是将不相关的文档排除。

should 过滤的数量是由 minimum_should_match 参数来进行控制，该参数可以是百分比，也可以是一个数字，我在多次实验后发现 40%的效果最好。

以下是 bool 的基本用法。

```
curl -XPOST '106.14.227.30: 9200/chageng/_search?pretty' -H 'Content-Type: application/json' -d '
{
"query": {
"bool": {
    "must": {
        "match": {
            "tagList": "游戏"
        }
    },
    "should": {
        "multi_match": {
            "query": "秦王",
            "type": "best_fields",
            "fields": [
                "name^5",
                  "tagList^2",
                "content^1"
            ],
            "tie_breaker": 0.4,
```

```
            "minimum_should_match": "40%"
        }
    },
    "filter": {
        "range": {
            "time": {
                "gte": "2012-09-09",
                "lt": "2019-09-09"
            }
        }
    }
}
}
}'
```

### 3.3 boost 权重控制

在多字段匹配中，我在 name tagList 和 content 字段中对内容进行查询，但是想让 name 字段拥有有更高的权重，可以通过指定 boost 来控制任何查询语句的相对的权重，boost 的默认值为 1，大于 1 会提升一个语句的相对权重。基本使用见上条 ES 语句。

基于 TF/IDF 的评分模型中，如果使用了 boost 改变权重，新的评分 _score 会在应用权重提升之后进行归一化处理，并不是线性的变化。

### 3.4 模糊匹配

模糊查询的工作原理是给定原始词项及构造一个编辑自动机— 像表示所有原始字符串指定编辑距离的字符串的一个大图表。然后模糊查询使用这个自动机依次高效遍历词典中的所有词项以确定是否匹配。 一旦收集了词典中存在的所有匹配项，就可以计算匹配

文档列表。在搜索巨大文档时候，模糊匹配的效率很低，故可以用以下两个参数限制对性能的影响，prefix*length* *为不能被“模糊化”的初始字符数，建议设置为了 3，* *max*expansions 限制产生的模糊选项的总数量。

```
curl -XPOST '106.14.227.30: 9200/chageng/_search?pretty' -H 'Content-Type: application/json' -d '
{
    "query": {
        "multi_match": {
            "query": "吓得我瓜子都掉了",
            "type": "best_fields",
            "fields": [
                "name^2",
                "content"
            ],
            "fuzziness": "AUTO",
            "tie_breaker": 0.4,
            "minimum_should_match": "40%"
        }
    }
}'
```

### 3.5 随机评分

我们的搜索词条结果集中有很多点赞数一样的词条，在指定按点赞数排序这种方式后，有相同评分 _score 的文档会每次都以相同次序出现，为了提高展现率，可以引入一些随机性，保证有相同评分的文档都能有均等相似的展现机率。

每个用户看到不同的随机次序，但也同时希望如果是同一用户翻页浏览时，结果的相对次序能始终保持一致。这种行为被称为 一致随机（consistently random）[1]。

以下是样例：

```
curl -XPOST '106.14.227.30: 9200/chageng/_search?pretty' -H 'Content-Type: application/json' -d '
{
  "query": {
    "function_score": {
      "filter": {
        "match": { "name": "小鸡快跑" }
      },
      "functions": [
        {
          "filter": { "term": { "tagList": "小鸡" }},
          "weight": 1
        },
        {
          "filter": { "term": { "tagList": "游戏" }},
          "weight": 2
        },
        {
          "random_score": {
            "seed":  "the users session id"
          }
        }
      ],
      "score_mode": "sum"
    }
  }
}
```

# 三、Elasticsearch 性能问题

## 1. 数据预热:

ES 可以在查询前进行预热，将查询中十分依赖的字段的数据加载出来，可以使用 Elasticsearch 为类型和索引定义预热查询[3]。

定义一个新的预热查询，和普通查询没什么区别，只是它存储在 Elasticsearch 一个特殊的名为_warmer 的索引中，以下是我的预热查询。

```
curl -XPUT '106.14.227.30: 9200/chageng/_warmer?pretty' -H 'Content-Type: application/json' -d '
{
    "query": {
        "match_all": {}
    },
    "facets": {
        "warming_facet": {
            "terms": {
                "field": "name"
            }
        }
    }
}'
```

## 2. 优化索引

我建立的索引，每个词条都包含较多的内容，不仅包括了该词条的基本信息（name，tag，content，view，like），还包括了从微博、B 站、谷歌等地方爬取到的相关信息，每个词条中包含的数据比较多，对完整的词条建立索引，每次的查询速度与在 mongodb 里面检索持平，在做自动补全时能够有肉眼可见的延迟。

出现查询速度过慢的情况有两方面的原因。第一是建立的索引包含的内容过多，比如微博、B 站的数据大约是该词条的基本信息的 25 倍以上，而这些微博、B 站的信息我们在做检索的时候并不需要对这些字段进行检索，这些无效的信息拖累了检索速度。第二是网络传输延迟，因为我们的 ES 集群和我们的搜索引擎服务并不在同一个机器上面，他们之间是通过网络进行通信，由于每条词条包含数据比较大，所以如果查询结果中有上百条的数据被命中，返回这些数据时需要比较多的时间。

我们的解决方案是建立两个索引，第一个索引存储词条的基本信息，第二个索引存储词条的所有信息，但使用检索功能的时候，我们只需要在第一个索引中检索，将词条的基本信息返回呈现给用户，这样可以大大加快速度。为了将速度优化到极致，在不影响用户正常使用的情况下，我们又对搜索结果的数量进行了限制，每次只返回当前页面要展现的搜索条目，最快的呈现给用户结果，其余的搜索结果用异步的方式加载。当用户进入词条详细页面时，我们可以通过该词条的 id，到 ES 中的第二个索引中查找该 id 词条的所有信息进行返回，这样的检索速度能够提升到 28 倍。

### 3. 优化存储

上文提到索引可以存储到 ES 中，这样的查询效率最高，那具体的数据可以存储到 MySQL、MongoDb，ES 三种数据库中，考虑到我们的数据类型以 json 文件为主，MySQL 需要建立多张表来实现关系间的映射，故不做考虑。

我对 MongoDb 存储词条的所有信息，与直接用 ES 存储所有的信息进行检索做了一个对比，发现两者在检索 10000 条数据运行时间上并没有太大差异，所以直接使用 ES 进行存储了所有的数据。

```
(base) [yryang:~]$
(base) [yryang:~]$ python test.py
<function query_in_es at 0x104be9a80>---run time--- 0.2602578639948291
<function query_in_mongodb at 0x104be9960>---run time--- 0.2348233933000291
```

图 2 MongoDb 与 Elasticsearch

## 参考文献:

[1] https://doc.yonyoucloud.com/doc/mastering-elasticsearch/chapter-3/31_README.html

[2] https://www.elastic.co/guide/cn/elasticsearch/guide/current/random-scoring.html

[3] https://doc.yonyoucloud.com/doc/mastering-elasticsearch/chapter-6/66_readme.html