Elasticsearch基础入门

|  |  |
| --- | --- |
| 编制 | 李聪8 |
| 审批 |  |

目录

[Elasticsearch基础入门 0](#_Toc483487046)

[1. 安装Elasticsearch 1](#_Toc483487047)

[2. 安装Kibana 2](#_Toc483487048)

[3. 安装X-Pack 2](#_Toc483487049)

[4. 安装Logstash 3](#_Toc483487050)

[5. Solr和ES对比 3](#_Toc483487051)

[6. Lucene基础知识 4](#_Toc483487052)

[6.1 名词理解 4](#_Toc483487053)

[6.2 索引与查询 4](#_Toc483487054)

[7. Elasticsearch基础知识 10](#_Toc483487055)

[7.1 Elasticsearch入门 10](#_Toc483487056)

[7.2 集群 13](#_Toc483487057)

[7.3 数据输入与输出 14](#_Toc483487058)

[7.4 分布式文档存储 18](#_Toc483487059)

[7.5 搜索 20](#_Toc483487060)

[7.6 映射和分析 21](#_Toc483487061)

[7.7 请求体查询 23](#_Toc483487062)

[7.8 排序与相关性 26](#_Toc483487063)

[7.9 分布式检索 27](#_Toc483487064)

[7.10 索引管理 28](#_Toc483487065)

[7.11 分片内部原理 32](#_Toc483487066)

[8. 深入搜索 37](#_Toc483487067)

[9. 数据建模 37](#_Toc483487068)

[10. 管理、监控和部署 37](#_Toc483487069)

[11. 修订记录 38](#_Toc483487070)

# 安装Elasticsearch

1. 安装JDK，略；
2. 下载elasticsearch的安装包，官网下载：

<https://www.elastic.co/products/elasticsearch>

1. 解压elasticsearch-5.3.1.tar.gz；
2. 创建用户；
   1. 创建用户组elasticsearch：groupadd elasticsearch
   2. 创建用户elasticsearch，并且加入到用户组elasticsearch中：

useradd -m elasticsearch -g elasticsearch

1. 给elasticsearch增加权限；

chown -R elasticsearch(用户):elasticsearch(用户组) elasticsearch(文件夹)

1. 基本参数配置
   1. 修改elasticsearch.yml参数配置文件：
      1. cluster.name: my-elasticsearch 集群名称；
      2. node.name: es1 节点名称；
      3. path.data: /path/to/data 数据目录；
      4. path.logs: /path/to/log 日志目录；
      5. bootstrap.system\_call\_filter: false 不调用系统过滤；
      6. network.host: 10.33.25.79 本机ip地址；
      7. http.port: 9200 本机端口，默认；
      8. discovery.zen.ping.unicast.hosts: ["node001"] 本机名称；
      9. discovery.zen.minimum\_master\_nodes: 3 防止脑裂参数；
      10. action.auto\_create\_index: False 不自动创建index；
2. 启动elasticsearch：
   1. 到elasticsearch解压的目录下执行：
      1. su - elasticsearch -c "${es\_home}/bin/elasticsearch -d"
      2. su - elasticsearch -c "${es\_home}/bin/elasticsearch -d –p ../elasticsearch.pid"
3. 关闭Elasticsearch的三种方式:
   1. Ctrl +C
   2. Kill 进程
   3. REST API

-XPOST http://localhost:9200/\_cluster/nodes/\_shutdown

–XPOST http://localhost:9200/\_cluster/nodes/node\_name/\_shutdown

1. FAQ
2. max file descriptors [65534] for elasticsearch process is too low, increase to at least [65536]
3. 修改/etc/security/limits.conf文件中最大文件数
4. max virtual memory areas vm.max\_map\_count [65530] is too low, increase to at least [262144]
5. 执行：sysctl -w vm.max\_map\_count=655360
6. 添加vm.max\_map\_count = 262144到/etc/sysctl.conf， 然后，sudo sysctl -p /etc/sysctl.conf
7. system call filters failed to install; check the logs and fix your configuration or disable system call filters at your own risk
8. 修改elasticsearch.yml中：bootstrap.system\_call\_filter: false

# 安装Kibana

Kibana is an open source analytics and visualization platform designed to work with Elasticsearch. You use Kibana to search, view, and interact with data stored in Elasticsearch indices. You can easily perform advanced data analysis and visualize your data in a variety of charts, tables, and maps.

1. 下载kibana

<https://www.elastic.co/downloads/kibana>

1. 基本配置
   1. 修改kibana.yml
      1. server.port: 5601
      2. server.host: "10.33.25.79"
      3. elasticsearch.url: "http://10.33.25.79:9200"

# 安装X-Pack

X-Pack is an Elastic Stack extension that bundles security, alerting, monitoring, reporting, and graph capabilities into one easy-to-install package. While the X-Pack components are designed to work together seamlessly, you can easily enable or disable the features you want to use.

1. 离线安装，下载X-Pack

<https://artifacts.elastic.co/downloads/packs/x-pack/x-pack-5.4.1.zip>

1. 安装X-Pack到elasticsearch中，在elasticsearch目录下执行命令：

bin/elasticsearch-plugin install file:///path/to/file/x-pack-5.3.1.zip

1. 安装X-Pack到kibana中，在kibana目录下执行命令：

bin/kibana-plugin install file:///path/to/file/x-pack-5.3.1.zip

1. 安装X-Pack到Logstash中，在logstash目录下执行命令：

bin/logstash-plugin install file:///path/to/file/x-pack-5.3.1.zip

1. 配置说明

|  |  |  |
| --- | --- | --- |
| 序号 | 设置 | 描述 |
| 1 | xpack.security.enabled | false：disable X-Pack security.  elasticsearch.yml and kibana.yml |
| 2 | xpack.monitoring.enabled | false：disable X-Pack monitoring.  elasticsearch.yml and kibana.yml |
| 3 | xpack.graph.enabled | false：disable X-Pack graph.  elasticsearch.yml and kibana.yml |
| 4 | xpack.watcher.enabled | false：disable X-Pack Watcher.  elasticsearch.yml |
| 5 | xpack.reporting.enabled | false：disable X-Pack reporting.  kibana.yml |

1. 默认的用户名和密码

用户：elastic 密码：changeme

# 安装Logstash

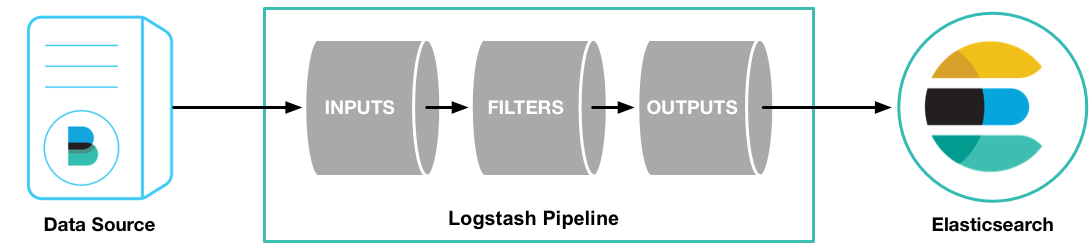
Logstash是一种分布式日志收集框架，开发语言是JRuby，当然是为了与[Java](http://lib.csdn.net/base/javase)平台对接，不过与Ruby语法兼容良好，非常简洁强大，经常与ElasticSearch，Kibana配置，组成著名的ELK技术栈，非常适合用来做日志数据的分析。

1. 下载Logstash

https://www.elastic.co/downloads/logstash

1. 解压tar.gz；
2. 配置（略）
3. demo：

bin/logstash -e 'input { stdin { } } output { stdout {} }'

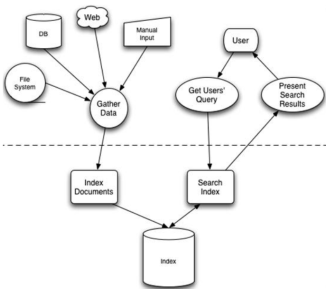


# Solr和ES对比

1. **查询方面**
   1. 对稳定数据的查询，Solr的速度略优于ES；
   2. 随着数据量的增加，Solr查询性能下降明显，而ES查询性能变化较小；
   3. 实时建索引时查询，ES秒杀Solr；Solr建索引产生IO堵塞，近实时查询性能较差；
2. **数据格式**
   1. Solr支持多种格式的数据，而ES仅支持json格式的数据；
3. **集群架构**
   1. Solr依赖于zookeeper搭建集群环境；Elasticsearch是分布式的，不依赖于其它组件；
   2. 在实际使用方面，ES在横向扩展方面优于Solr，扩容更简单；Solr扩容稍微复杂些；
4. **其它**
   1. ES更加流行，Solr更加稳定；
   2. Solr是传统搜索应用的有力解决方案，比如电商；而ES 更适用于新兴的实时搜索应用，比如新浪微博；
   3. Solr官方提供的功能较多；ES自身更注重于核心功能，高级功能依赖于第三方插件；ES栈（比如ELK）功能强大，模块化插拔，更受欢迎；

# Lucene基础知识

Lucene 是apache的一个开源全文检索引擎工具包，也是一个全文检索引擎的架构，提供了完整的查询引擎和索引引擎，部分文本分析引擎。如下图**全文检索**的过程:



## 名词理解

1. **全文检索**

对非结构化的数据搜索主要有两种方法：

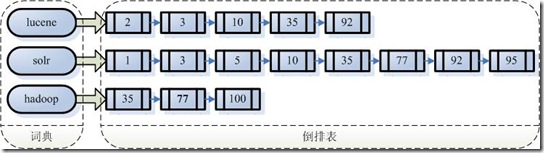
（1）顺序扫描法：Linux下的grep命令；Windows中的文件扫描；

**（2）全文检索：**先建立索引，再对索引进行搜索的过程；

1. **索引**：从非结构化数据中提取并重新组织的结构化数据，用于提高数据检索的效率；（名词，动词）
2. **倒排索引**：也叫反向索引，是关键词到文档的一种映射关系；

## 索引与查询

假设文档集合里面有100篇文档，为了方便表示，我们为文档编号从1到100，得到下面的结构：



左边保存的是字符串叫做**字典表**；

每个字符串指向包含此字符串的文档链表，此文档链表称为**倒排表** 。

**举例**：搜索solr and lucene

* + 1. 取出包含字符串“lucene”的文档链表；
    2. 取出包含字符串“solr”的文档链表；
    3. 合并链表，找出既包含“lucene”又包含“solr”的文件。

[图]倒排表合并过程

1. **创建索引步骤**
2. **准备好文档集合---（Document）**

Doc1：Students should be allowed to [Go](http://lib.csdn.net/base/go) out with their friends, but not allowed to drink beer.

Doc2：My friend Jerry went to school to see his students but found them drunk which is not allowed.

1. **对文档进行分词---分词器（Tokenizer）**

分词器对文档进行切分单词、去除标点符号、去除停用词（不同语言可自定义停用词集合）等；最后得到的结果叫做词元（Token）。

“Students”，“allowed”，“go”，“their”，“friends”，“allowed”，“drink”，“beer”，“My”，“friend”，“Jerry”，“went”，“school”，“see”，“his”，“students”，“found”，“them”，“drunk”，“allowed”。

1. **语言处理组件进一步处理词元（Linguistic Processor）**

语言处理组件（LP）改变词元的大小写、提取词元的词根等；提取词根有两种方式：一是如复数变单数，cars—car，这种方式叫做stemming；二是变形的恢复到词根，drove—drive,这种方式叫做lemmatization**。**二者区别在于方式和算法不同。

语言处理组件处理的结果成为**词**（Term）。

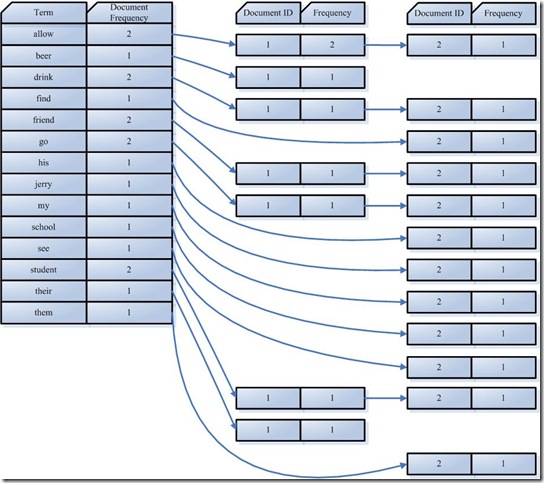
“student”，“allow”，“go”，“their”，“friend”，“allow”，“drink”，“beer”，“my”，“friend”，“jerry”，“go”，“school”，“see”，“his”，“student”，“find”，“them”，“drink”，“allow”。

1. **索引组件对词构建倒排索引（Indexer）**

**首先**，索引组件通过词创建一个字典表，并且按照字母顺序排序：

|  |  |
| --- | --- |
| **Term** | **Document ID** |
| allow | 1 |
| allow | 1 |
| allow | 2 |
| beer | 1 |
| drink | 1 |
| drink | 2 |
| find | 2 |
| friend | 1 |
| friend | 2 |
| go | 1 |
| go | 2 |
| his | 2 |
| jerry | 2 |
| my | 2 |
| school | 2 |
| see | 2 |
| student | 1 |
| student | 2 |
| their | 1 |
| them | 2 |

**其次，**合并相同的词(Term) 成为文档倒排(Posting List) 链表；



Document Frequency 即文档频次，表示总共有多少文件包含此词(Term)；

Frequency 即词频率，表示此文件中包含了几个此词(Term)。

**最后**，合并以及压缩；

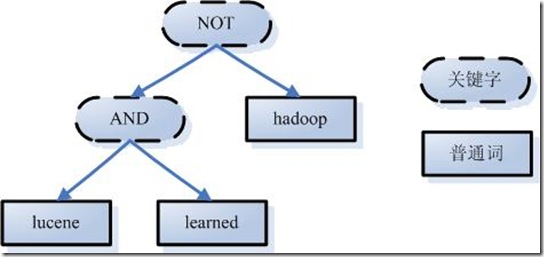
1. **查询的步骤**
2. **用户输入查询语句**

例如：用户输入语句：lucene AND learned NOT [Hadoop](http://lib.csdn.net/base/hadoop)；

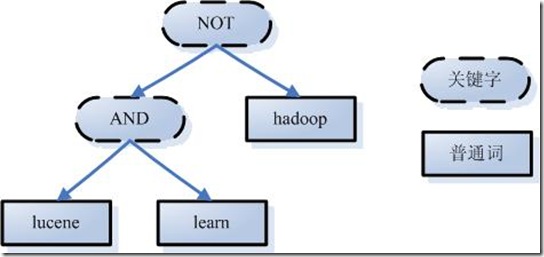
1. **对查询语句进行词法分析、语法分析和语言处理**

**首先**，通过分词得到关键词和关键字：lucene，learned，hadoop, 关键字有AND, NOT；

**其次**，生成语法树；



**最后**，语言处理，与索引过程相同；如learned---learn



1. **搜索索引，得到符合语法树的文档**

**首先**，分别在倒排索引中找到包含lucene，learn，hadoop的文档链接；

**其次**，合并lucene和learn的结果；

**然后**，将上一步中的结果对hadoop的结果做差，去除hadoop的链接；

**最后**，上一步中的文档链接就是要找的文档；

1. **计算文档与查询结果之间的相关性，并且排序**

方法1：**向量空间模型的算法(Vector Space Model)**

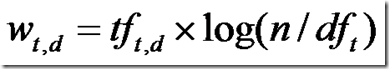
把输入的查询语句看作是一个小文档，目标：计算小文档与查询结果中的文档的相关性；

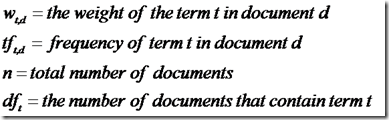
**首先**，计算词的权重，即找到词（Term）对文档的重要程度，越重要的词有越大的权重(Term weight)；

影响一个词在一篇文档中的重要性主要有两个因素：

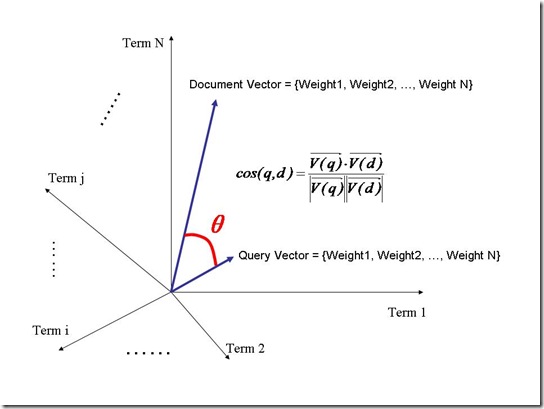
Term Frequency (tf)：该Term在该文档中出现次数，tf 越大说明越重要；

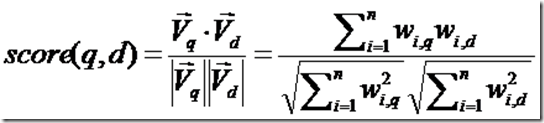
Document Frequency (df)：包含该Term的文档数，df 越大说明越不重要。





**其次**，根据权重向量计算文档之间的相似度；





**例子**：查询语句有11个Term，共有三篇文档搜索出来

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 | t10 | t11 |
| D1 | 0 | 0 | .477 | 0 | .477 | .176 | 0 | 0 | 0 | .176 | 0 |
| D2 | 0 | .176 | 0 | .477 | 0 | 0 | 0 | 0 | .954 | 0 | .176 |
| D3 | 0 | .176 | 0 | 0 | 0 | .176 | 0 | 0 | 0 | .176 | .176 |
| Q | 0 | 0 | 0 | 0 | 0 | .176 | 0 | 0 | .477 | 0 | .176 |

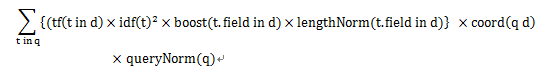
计算三篇文档同查询语句的相关性打分分别为：

[图]文档一的打分计算

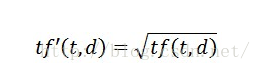
[图]文档二的打分计算

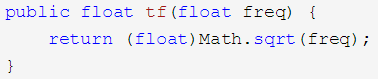
[图]文档三的打分计算

方法2：**Lucene的评分机制**

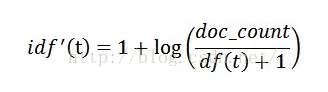


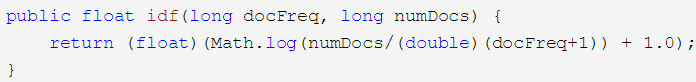
* + - 1. tf(t in d)：项t在文档d中出现的次数





* + - 1. idf(t) ：逆文档频率, 包含项t的文档数



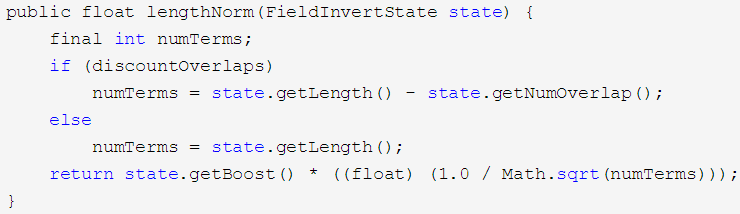


* + - 1. boost(t,field in d)：查询时查询项加权，在索引时给域和文档的加权

**t.getBoost()**

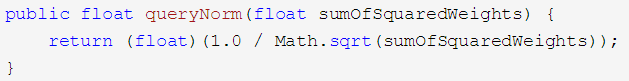
* + - 1. lengthNorm(t.field in d)：长度相关的加权因子

目的是为了将同样匹配的文档，比较短的放比较前面；

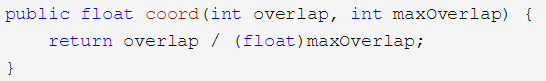


* + - 1. queryNorm(q)：每查询的归一化值

这个因素对所有文档都是一样的值，所以它不影响排序结果。比如如果我们希望所有文档的评分大一点，那么我们就需要设置这个值。



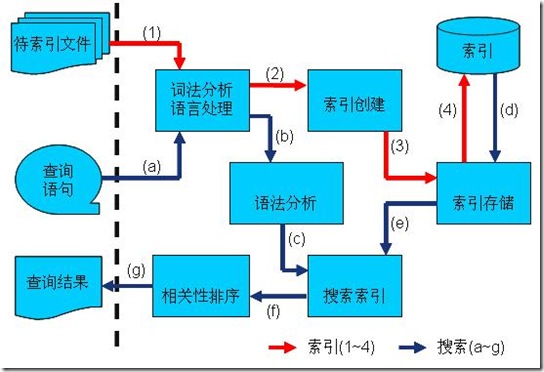
* + - 1. coord(q,d)： 评分因子



overlap: 文档中命中检索的个数

maxOverlap: 检索条件的个数

1. **将排序后的结果返回给用户**
2. **Java-Demo**
   * + 1. 构建索引与查询；
3. **总结**



# Elasticsearch基础知识

*Elasticsearch* 是一个建立在Lucene基础之上的开源的、实时的分布式搜索分析引擎。它被用作全文检索、结构化搜索、分析以及这三个功能的组合：

1. 一个分布式的实时文档存储，每个字段可以被索引与搜索；
2. 一个分布式实时分析搜索引擎；
3. 能胜任上百个服务节点的扩展，并支持 PB 级别的结构化或者非结构化数据。

## Elasticsearch入门

1. **Java API**

Elasticsearch 内置的两个JAVA客户端：

**节点客户端**（Node client）：作为一个非数据节点加入到本地集群中，可以把请求转发到其它节点；

**传输客户端**（Transport client）：它可以将请求发送到远程集群。

两者均是通过9300端口与集群通信。

1. **RESTFULL API with JSON over HTTP**

Elasticsearch 请求：

curl -X<VERB> '<PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY\_STRING>' -d '<BODY>'

* 1. **VERB**：HTTP方法或谓词 : GET`、`POST`、`PUT`、`HEAD 或 `DELETE`；
  2. **PROTOCOL**：http 或者 https；
  3. **HOST**：Elasticsearch集群中任意节点的主机名，或者用 localhost 代表本地机器上的节点；
  4. **PORT**：运行Elasticsearch HTTP服务的端口号，默认是 9200；
  5. **PATH**：API 的终端路径（例如 \_count 将返回集群中文档数量），Path 可能包含多个组件，例如：\_cluster/stats；
  6. **QUERY\_STRING**：任意可选的查询字符串参数，例如 ?pretty 将格式化地输出 JSON 返回值；
  7. **BODY**：一个JSON格式的请求体。

1. **索引与搜索Demo**
   1. **构建索引**

PUT /megacorp/employee/1{

"first\_name" : "John",

"last\_name" : "Smith",

"age" : 25,

"about" : "I love to go rock climbing",

"interests": [ "sports", "music" ]

}

**megacorp：**索引名称； **employee：**类型名称；

关系型数据库、Elasticsearch和Solr三者的对比：

Relational DB -> Databases -> Tables -> Rows -> Columns

Elasticsearch -> Indices -> Types -> Documents -> Fields

Solr -> index -> core -> Documents -> Fields

* 1. **搜索文档**

根据需要采用不同的搜索：简单搜索、复杂搜索、全文搜索、短语搜索、高亮搜索等；

* + 1. **简单搜索**

GET /megacorp/employee/1

GET /megacorp/employee/\_search

GET /megacorp/employee/\_search?q=last\_name:Smith

* + 1. **复杂搜索**

GET /megacorp/employee/\_search {

"query" : {

"match" : {

"last\_name" : "Smith"

}

}

}

GET /megacorp/employee/\_search {

"query" : {

"bool": {

"must": {

"match" : {

"last\_name" : "smith" https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

}

},

"filter": {

"range" : {

"age" : { "gt" : 30 } https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/2.png

}

}

}

}

}

* + 1. **全文搜索**

GET /megacorp/employee/\_search

{

"query" : {

"match" : {

"about" : "rock climbing"

}

}

}

* + 1. **短语搜索**

同时包含rock和climbing,且两个单词作为一个短语

GET /megacorp/employee/\_search

{

"query" : {

"match\_phrase" : {

"about" : "**rock climbing**"

}

}

}

* + 1. **高亮搜搜**

GET /megacorp/employee/\_search

{

"query" : {

"match\_phrase" : {

"about" : "rock climbing"

}

},

"highlight": {

"fields" : {

"about" : {}

}

}

}

1. **分析**

GET /megacorp/employee/\_search

{

"aggs": {

"all\_interests": {

"terms": { "field": "interests" }

}

}

}

## 集群

1. **集群健康状态**
   1. **green 、 yellow 或者 red**

green: 所有的主分片和副本分片都正常运行;

yellow ：所有的主分片都正常运行，但不是所有的副本分片都正常运行;

red ：有主分片没能正常运行。

GET /\_cluster/health

1. **分片**

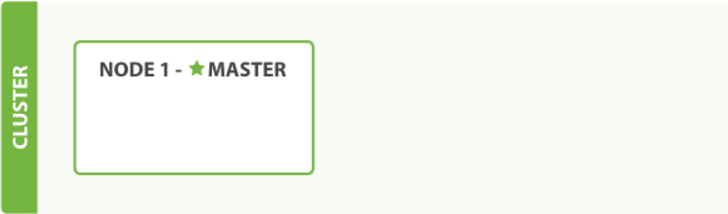
索引实际上是指向一个或者多个物理分片的逻辑命名空间 ；分片是数据的容器，文档保存在分片内，分片又被分配到集群内的各个节点里，分片是一个功能完整的搜索引擎。当你的集群规模扩大或者缩小时，Elasticsearch会自动的在各节点中迁移分片，使得数据仍然均匀分布在集群里。

分片可以是主分片或者副本分片（个数可修改），主分片的数目决定着索引能够保存的最大数据量。 Integer.MAX\_VALUE - 128 （参考值）

1. **横向扩容与容灾**

**横向扩容：**

* 1. 没有填加分片的集群，及空集群，康状态是green的；单机为例：



* 1. 向该单机空集群添加分片；集群状态变为yellow；

PUT /blogs {

"settings" : {

"number\_of\_shards" : 3,

"number\_of\_replicas" : 1

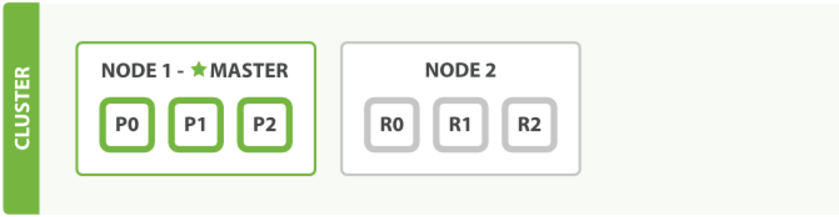
}

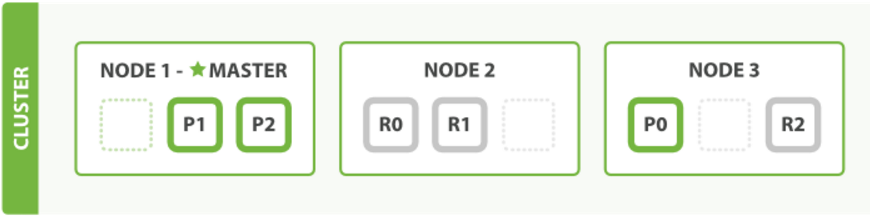
}



* 1. 向该集群加入2个节点；集群健康状态变为green；

**添加方法：**修改被添加的节点的集群名称与NODE集群名称相同；





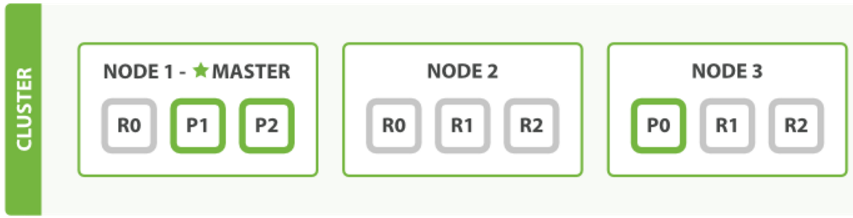
**容灾：**

1. 三个节点，修改副本分片数为2：

PUT /blogs/\_settings {

"number\_of\_replicas" : 2

}



1. 关掉NODE-1，集群状态变为yellow，瞬间变为green；重新选择主分片；



## 数据输入与输出

重点： Elasticsearch 中怎样安全的存储文档，以及如何将文档再次返回。

Elasticsearch的存储和检索的数据结构：序列化成为JSON文档；

1. **文档元数据**

\_index：文档存放在哪

\_type：文档表示的对象类别

\_id：文档唯一标识

1. **操作文档**

GET、POST、PUT、DELETE和HEAD等

* 1. **索引文档**

PUT /{index}/{type}/{id}

{

"field": "value",

...

}

自定义ID或者自动生成ID；

* 1. **取回一个文档**
     1. 返回文档的全部信息:

GET /website/blog/123?pretty

* + 1. 返回文档一部分:

GET /website/blog/123?\_source=title,text

GET /website/blog/123/\_source

* 1. **检查文档是否存在**

curl -i -XHEAD http://localhost:9200/website/blog/123

* 1. **更新文档**
     1. **更新整个文档**

在Elasticsearch中文档是***不可改变*** 的，不能修改它们。

PUT /website/blog/123

{

"title": "My first blog entry",

"text": "I am starting to get the hang of this...",

"date": "2014/01/02"

}

在响应体中，我们能看到 Elasticsearch 已经增加了 \_version 字段值：

* + 1. **文档的部分更新**

第一种方式：doc

POST /website/blog/1/\_update

{

"doc" : {

"tags" : [ "testing" ],

"views": 0

}

}

第二种方式：script

POST /website/blog/1/\_update

{

"script" : "ctx.\_source.tags+=new\_tag",

"params" : {

"new\_tag" : "search"

}

}

第三种方式：upsert，如果不存在，则先创建；

POST /website/pageviews/1/\_update

{

"script" : "ctx.\_source.views+=1",

"upsert": {

"views": 1

}

}

* 1. **创建新索引**

如果文档已经存在，则不创建索引；反之，创建索引。两种方式：

* + 1. **第一种方法：**使用 op\_type 查询 -字符串参数：

PUT /website/blog/123?op\_type=create

{ ... }

* + 1. **第二种方法：**是在 URL 末端使用 /\_create：

PUT /website/blog/123/\_create

{ ... }

返回值：201，Created；409，Conflict。

* 1. **删除文档**

DELETE /website/blog/123

删除文档不会立即将文档从磁盘中删除，只是将文档标记为已删除状态.

* 1. **处理冲突**
     1. 悲观并发控制；锁住；
     2. 乐观并发控制；系统版本号和外部版本号
  2. **取回多个文档**

GET /\_mget

{

"docs" : [

{

"\_index" : "website",

"\_type" : "blog",

"\_id" : 2

},

{

"\_index" : "website",

"\_type" : "pageviews",

"\_id" : 1,

"\_source": "views"

}

]

}

docs中的type覆盖API中的type;

GET /website/blog/\_mget

{

"docs" : [

{ "\_id" : 2 },

{ "\_type" : "pageviews", "\_id" : 1 }

]

}

* 1. **批量操作**

{ action: { metadata }}\n

{ request body }\n

{ action: { metadata }}\n

{ request body }\n

...

action：

create： 如果文档不存在，那么就创建它；

index：创建一个新文档或者替换一个现有的文档；

update：部分更新一个文档；

delete：删除一个文档；

metadata

 \_index 、 \_type 和 \_id；

POST /\_bulk

{ "delete": { "\_index": "website", "\_type": "blog", "\_id": "123" }}

{ "create": { "\_index": "website", "\_type": "blog", "\_id": "123" }}

{ "title": "My first blog post" }

{ "index": { "\_index": "website", "\_type": "blog" }}

{ "title": "My second blog post" }

{ "update": { "\_index": "website", "\_type": "blog", "\_id": "123", "\_retry\_on\_conflict" : 3} }

{ "doc" : {"title" : "My updated blog post"} }

问题：这里为什么不使用json请求体？

Elasticsearch可以直接读取被网络缓冲区接收的原始数据。它使用换行符字符来识别和解析小的 action/metadata 行来决定哪个分片应该处理每个请求。

这些原始请求会被直接转发到正确的分片。没有冗余的数据复制，没有浪费的数据结构。整个请求尽可能在最小的内存中处理。

总结：

1. GET：获取请求对象的当前状态；
2. POST：改变对象的当前状态；
3. PUT：创建一个对象；
4. DELETE：删除一个对象；
5. HEAD：获取对象的基本信息；

## 分布式文档存储

1. **路由一个文档到一个分片中**

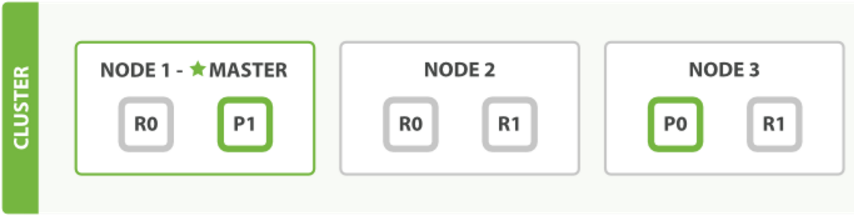
公式：shard = hash(routing) % number\_of\_primary\_shards

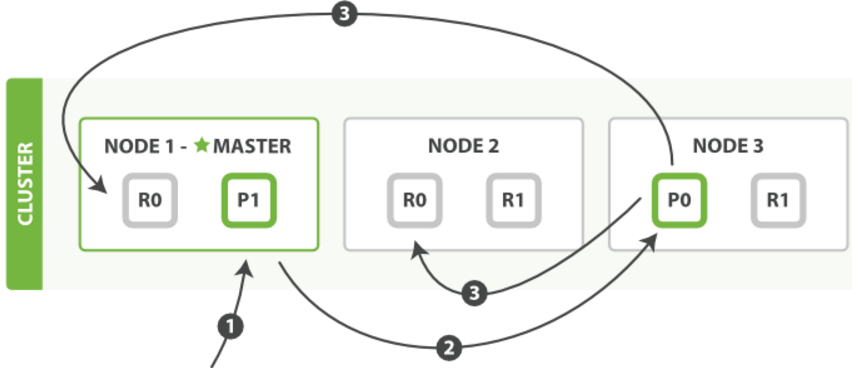
hash：一个hash函数；

routing：是一个可变值，默认是文档的 \_id ，也可自定义；

number\_of\_primary\_shards：主分片的数量。

1. **新建、索引和删除文档**





* 1. 客户端向 Node 1 发送新建、索引或者删除请求；
  2. 节点使用文档的 \_id 确定文档属于分片 0，请求会被转发到 `Node 3`。
  3. Node 3 在主分片上面执行请求。如果成功了，它将请求并行转发到 Node 1 和 Node 2 的副本分片上。一旦所有的副本分片都报告成功, Node 3 将向协调节点报告成功，协调节点向客户端报告成功。

**一致性**

默认情况下，主分片需要规定数量或大多数的分片在写入操作时可用。

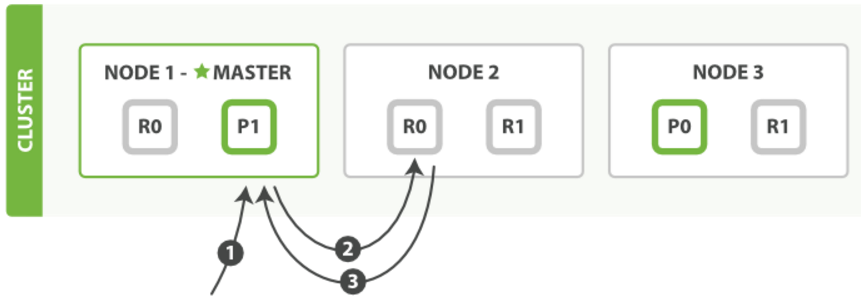
int( (primary + number\_of\_replicas) / 2 ) + 1

number\_of\_replicas： 索引中设置指定的分片数；

**超时**

timeout：默认一分钟，可设置；

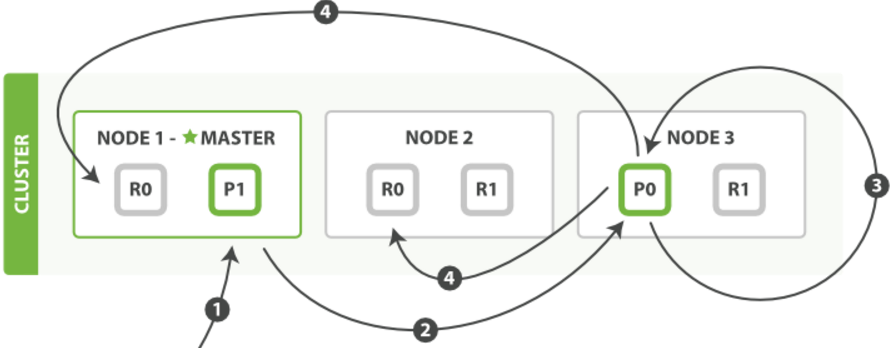
1. **取回一个文档**



* 1. 客户端向 Node 1 发送获取请求；
  2. 节点使用文档的 \_id 来确定文档属于分片 0 。分片 0 的副本分片存在于所有的三个节点上。在这种情况下，它将请求转发到 Node 2 ；
  3. Node 2 将文档返回给 Node 1 ，然后将文档返回给客户端。

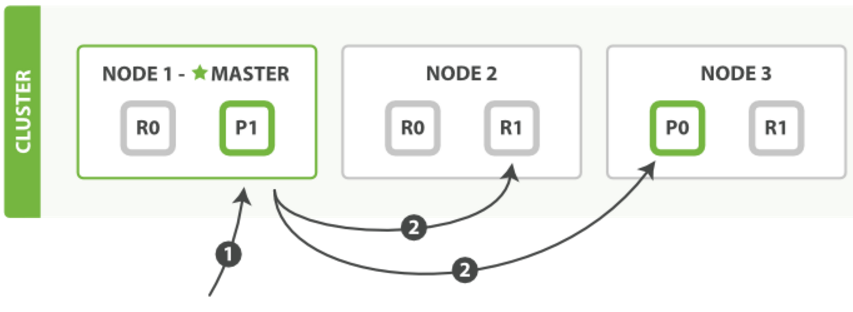
问题：如果建索引时检索文档，且主分片上已经存在，但是副本分片上还没有复制成功，而检索请求刚好发送到副本分片上，那返回结果会不会有误？

1. **局部更新文档**

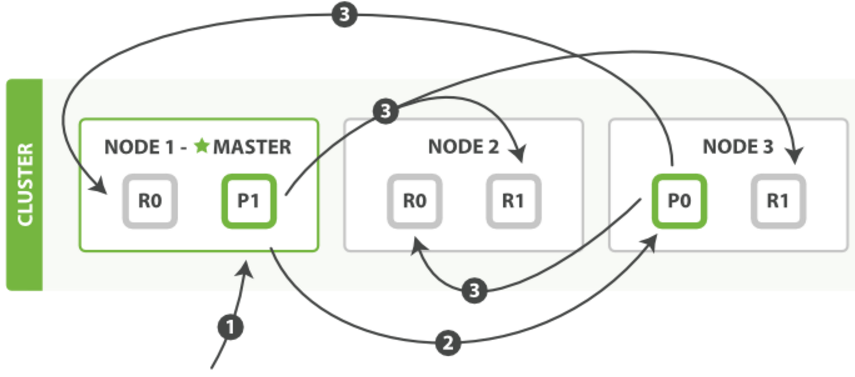


* 1. 客户端向 Node 1 发送更新请求；
  2. 它将请求转发到主分片所在的 Node 3 ；
  3. Node 3 从主分片检索文档，修改 \_source 字段中的 JSON ，并且尝试重新索引主分片的文档。如果文档已经被另一个进程修改，它会重试步骤 3 ，超过 retry\_on\_conflict 次后放弃；
  4. 如果 Node 3 成功地更新文档，它将新版本的文档并行转发到 Node 1 和 Node 2 上的副本分片，重新建立索引。 一旦所有副本分片都返回成功， Node 3 向协调节点也返回成功，协调节点向客户端返回成功。

1. **多文档模式**
   1. **mget取回多个文档**



* 1. 客户端向 Node 1 发送 mget 请求;
  2. Node 1 为每个分片构建多文档获取请求，然后并行转发这些请求到托管在每个所需的主分片或者副本分片的节点上。一旦收到所有答复， Node 1 构建响应并将其返回给客户端。
  3. **Bulk修改多个文档**



1. 客户端向 Node 1 发送 bulk 请求；
2. Node 1 为每个节点创建一个批量请求，并将这些请求并行转发到每个包含主分片的节点主机；
3. 主分片一个接一个按顺序执行每个操作。当每个操作成功时，主分片并行转发新文档（或删除）到副本分片，然后执行下一个操作。一旦所有的副本分片报告所有操作成功，该节点将向协调节点报告成功，协调节点将这些响应收集整理并返回给客户端。

## 搜索

1. **空搜索**

搜索API的最基础的形式是没有指定任何查询的空搜索：

GET /\_search

返回结果：

hits：包含 total 字段来表示匹配到的文档总数，并且一个 hits 数组包含所查询结果的前十个文档；

\_score： 衡量了文档与查询的匹配程度；

max\_score：与查询所匹配文档的 \_score 的最大值；

took：整个搜索请求耗费了多少毫秒；

\_shards：查询中参与分片的总数，以及这些分片成功了多少个失败了多少个；

timed\_out：查询是否超时；指定超时时间：GET /\_search?timeout=10ms；

1. **多索引多类型**

在一个或多个特殊的索引并且在一个或者多个特殊的类型中进行搜索：

/\_search：在所有的索引中搜索所有的类型；

/gb/\_search：在 gb 索引中搜索所有的类型；

/gb,us/\_search：在 gb 和 us 索引中搜索所有的文档；

/g\*,u\*/\_search：在任何以 g 或者 u 开头的索引中搜索所有的类型；

/gb/user/\_search：在 gb 索引中搜索 user 类型；

/gb,us/user,tweet/\_search：在 gb 和 us 索引中搜索 user 和 tweet 类型；

/\_all/user,tweet/\_search：在所有的索引中搜索 user 和 tweet 类型；

1. **分页**

size：显示应该返回的结果数量，默认是10；

from：显示应该跳过的初始结果数量，即offset，默认是 0；

GET /\_search?size=5&from=10

深度分页问题：

假设在一个有5个主分片的索引中搜索，我们请求第1000页--结果从10001 到10010；每个分片都会产生前10010个结果，然后，协调节点对全部的5\*10010个结果排序；

所以，在分布式的系统中，对结果排序的成本是随分页的深度成指数上升。

1. **轻量搜索**

搜索分为轻量搜索(查询字符串 版本，要求在查询字符串中传递所有参数)和完整的请求体搜索（JSON请求体或更丰富的表达式）；

GET /\_all/tweet/\_search?q=tweet:elasticsearch+name:john

+: 前缀表示必须与查询条件匹配；

-：前缀表示一定不与查询条件匹配；

无：所有其他条件都是可选的——匹配的越多，文档就越相关。

\_all**字段**：当索引一个文档的时候，Elasticsearch取出所有字段的值拼接成一个大的字符串，作为 \_all 字段进行索引。

## 映射和分析

GET /gb/\_mapping/tweet

1. **分析**

Elasticsearch中的数据可以分为两类：精确值和全文

参见6 Lucene基础知识一节；

分析：分词和标准化的过程；

内置分析器：标准分析器，简单分析器，空格分析器，语言分析器等。

GET /\_analyze

{

"analyzer": "standard",

"text": "Text to analyze"

}

1. **映射**

映射定义了类型中的域，每个域的数据类型，以及Elasticsearch如何处理这些域。映射也用于配置与类型有关的元数据。

1. **域类型**

**核心简单域类型：**

字符串： string

整数：byte, short, integer, long

浮点数：float, double

布尔型：Boolean

日期：date

如果没有建立映射表，Elasticsearch会使用动态映射，通过JSON中基本的数据类型，尝试猜测域类型；

**复杂核心域类型：**

Elasticsearch中的JSON支持null值，数组和对象。

**多值域：**

{ "tag": [ "search", "nosql" ]}

数组中所有的值必须是相同数据类型的，索引和搜索时是无序的。

**空域：**

"null\_value": null,

"empty\_array": [],

"array\_with\_null\_value": [ null ]

Lucene不支持空值的。

**多层级对象：**

  JSON 原生数据类是对象--其他语言中称为哈希 map或关联数组；

内部对象经常用于嵌入一个实体或对象到其它对象中；

内部对象映射：

内部对象如何建索引：

Lucene不支持复杂的数据结构，Elascticsearch做了扁平化处理：

{

"tweet": [elasticsearch, flexible, very],

"user.id": [@johnsmith],

"user.gender": [male],

"user.age": [26],

"user.name.full": [john, smith],

"user.name.first": [john],

"user.name.last": [smith]

}

内部对象数组：

{

"followers": [

{ "age": 35, "name": "Mary White"},

{ "age": 26, "name": "Alex Jones"},

{ "age": 19, "name": "Lisa Smith"}

]

}

1. **自定义域映射**
2. 对于不是string的域，一般化只需要设置type就OK了。
3. string类型域会被认为包含全文，有2个重要的属性：index和analyzer；

index属性值：

analyzed：首先分析字符串，然后，索引它；

not\_analyzed：索引该域，可搜索它，不对它分析，索引指定精确值。

no：不索引这个域，这个域不会被搜索到。

analyzer属性值：

对analyzed字符串域，analyzer属性指定搜索和索引时使用分析器；

1. **更新映射**

更新一个映射来添加一个新域，但不能将一个存在的域从 analyzed 改为 not\_analyzed。

## 请求体查询

1. **空查询**

GET /\_search

{}

1. **查询表达式**

{

QUERY\_NAME: {

ARGUMENT: VALUE,

ARGUMENT: VALUE,...

}

}

叶子语句：被用于将查询字符串和一个字段（或者多个字段）对比；

复合语句：主要用于合并其它查询语句。

1. **查询与过滤**

Elasticsearch 使用的查询语言（DSL） 拥有一套可以以无限组合的方式进行搭配的查询组件，这套组件有两种情况下：**过滤情况**和**查询情况**。

**过滤情况：**

查询被设置成一个“不评分”或者“过滤”查询。即这个查询只是简单的问一个问题：“这篇文档是否匹配？”，回答yes或no。

**查询情况：**

查询就变成了一个“评分”的查询。它不仅要去判断这个文档是否匹配，同时还要判断这个文档匹配的有多好（匹配程度如何，\_score）。

**两者性能差异：**

过滤查询只是简单的检查包含或者排除，计算非常快，结果会被缓存到内存中以便快速读取。

评分查询不仅要找出匹配的文档，还要计算每个匹配文档的相关性，计算相关性使得它们比不评分查询费力的多。同时，查询结果并不缓存。

过滤的目标是减少那些需要通过评分查询进行检查的文档。

1. **最重要查询**

**match\_all查询：**查询简单的匹配所有文档，默认的查询；

{ "match\_all": {}}

**match 查询：**标准查询，用于精确或全文搜索；

{ "match": { "tweet": "About Search" }}

**multi\_match 查询：**在多个字段上执行相同的 match 查询；

{

"multi\_match": {

"query": "full text search",

"fields": [ "title", "body" ]

}

}

**range 查询：**指定区间内的数字或者时间；

{

"range": {

"age": {

"gte": 20,

"lt": 30

}

}

}

gt：大于； gte：大于等于； lt：小于； lte：小于等于

**term 查询：**用于**精确值**匹配，精确值可能是数字、时间、布尔或 not\_analyzed 的字符串；

{ "term": { "age": 26 }}

{ "term": { "date": "2014-09-01" }}

{ "term": { "public": true }}

{ "term": { "tag": "full\_text" }}

term 查询对于输入的文本**不分析** ，所以它将给定的值进行精确查询。

**terms 查询：**允许你指定多值进行匹配，对于输入的文本**不分析；**

{ "terms": { "tag": [ "search", "full\_text", "nosql" ] }}

term 和 terms 是 *包含（contains）* 操作，而非 *等值（equals）*（判断）。

**exists 查询和 missing 查询：**用于查找指定的字段中有值 (exists) 或无值 (missing) 的文档；

{

"exists": {

"field": "title"

}

}

这些查询经常用于某个字段有值的情况和某个字段缺值的情况。

1. **组合多查询**

在多个字段上查询多种多样的文本，并且根据一系列的标准来过滤。为了构建类似的高级查询，需要一种能够将多查询组合成单一查询的查询方法。**bool**查询可以实现这种需求。

参数：

must：文档**必须** 匹配这些条件才能被包含进来；AND

must\_not：文档**必须不**匹配这些条件才能被包含进来；NOT

should：如果满足语句中**任意**语句，将增加 \_score ，否则，无任何影响；OR

filter：必须 匹配，但它以不评分、过滤模式来进行。

通过多个不同的标准来过滤文档，bool 查询本身也可以被用做不评分的查询，将bool放在filter之中即可。

constant\_score 查询：它将一个不变的常量评分应用于所有匹配的文档。可以使用它来取代只有 filter 语句的 bool 查询。

{

"constant\_score": {

"filter": {

"term": { "category": "ebooks" }

}

}

}

1. **验证查询**

validate-query API 可以用来验证查询是否合法；explain找出查询不合法的原因。

GET /gb/tweet/\_validate/query?explain

{

"query": {

"tweet" : {

"match" : "really powerful"

}

}

}

## 排序与相关性

1. **按照字段的值排序**

GET /\_search{

"query" : {

"bool" : {

"filter" : { "term" : { "user\_id" : 1 }}

}

},

"sort": { "date": { "order": "desc" }}

}

也可以指定一个字段排序，默认升序排序：

"sort": "number\_of\_children"

1. **多级排序**

"sort": [

{ "date": { "order": "desc" }},

{ "\_score": { "order": "desc" }}

]

结果首先按第一个条件排序，仅当结果集的第一个 sort 值完全相同时才会按照第二个条件进行排序，以此类推。

1. **字段多值排序**

对于数字或日期，可以使用min，max，avg或sum等排序；

"sort": {

"dates": {

"order": "asc",

"mode": "min"

}}

1. **字符串排序与多字段**

传递一个**单字段**但是却用两种方式索引它。修改映射，添加一个field：

"tweet": {

"type": "string",

"analyzer": "english",

"fields": {

"raw": {

"type": "string",

"index": "not\_analyzed"

}}}

排序的时候，用tweet.raw；

"sort": "tweet.raw"

1. **相关性**

Elasticsearch的相似度算法 被定义为检索词频率/反向文档频率， *TF/IDF* ，包括以下内容：

检索词频率

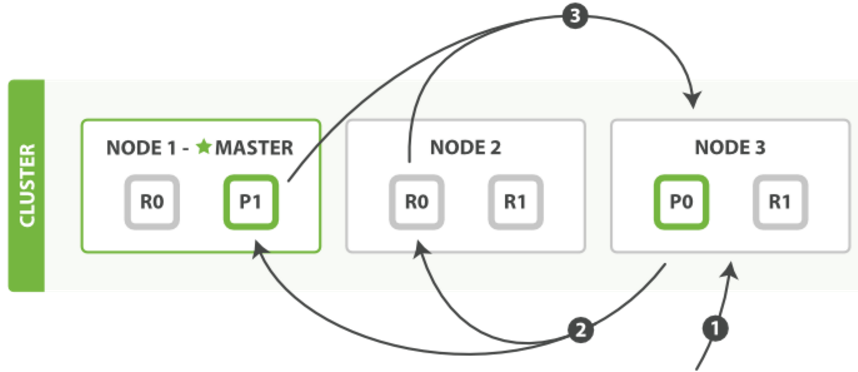
反向文档频率

字段长度准则：长度越长，相关性越低；

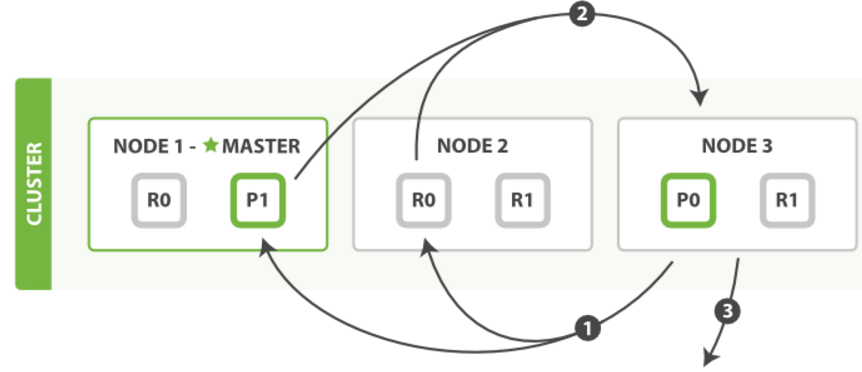
## 分布式检索

搜索分为两个阶段：query和fetch。

1. **查询阶段**



1. **发送请求，创建本地优先队列：** 客户端发送search 请求到 Node 3 ，Node 3 创建一个大小为 from + size 的空优先队列；
2. **转发请求，创建本地优先队列：**Node 3 将查询请求转发到索引的每个主分片或副本分片中。每个分片在本地执行查询并添加结果到大小为 from + size 的本地有序优先队列中；
3. **合并分片结果，重排序：**每个分片返回各自优先队列中所有文档的ID和排序值给协调节点，也就是 Node 3 ，它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表；
4. **取回阶段**
   1. 协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求；
   2. 每个分片加载并丰富文档，如果有需要的话，接着返回文档给协调节点；
   3. 一旦所有的文档都被取回了，协调节点返回结果给客户端。



1. **搜索选项**
   1. 偏好：控制由哪些分片或节点来处理搜索请求；
   2. 超时：分片允许处理数据的最大时间；
   3. 路由：通过索引时定时参数routing，在搜索时，指定搜索几个相关的分片；

GET /\_search?routing=user\_1,user2

* 1. 搜索类型：默认的搜索类型是query\_then\_fetch；（后面讨论）

GET /\_search?search\_type=dfs\_query\_then\_fetch

1. **游标查询（略）**

scroll 查询：游标查询先做查询初始化，然后，再批量地拉取结果；可以用来对 Elasticsearch有效地执行大批量的文档查询，而又不用付出深度分页那种代价。

## 索引管理

1. **创建一个索引**

首先，传入设置和类型映射：

PUT /my\_index

{

"settings": { ... any settings ... },

"mappings": {

"type\_one": { ... any mappings ... },

"type\_two": { ... any mappings ... },

...

}

}

可以在 config/elasticsearch.yml禁止自动创建索引：

action.auto\_create\_index: false

1. **删除一个索引**
   1. DELETE /my\_index
   2. DELETE /index\_one,index\_two
   3. DELETE /index\_\*
   4. DELETE /\_all
   5. DELETE /\*

为了避免删除所有索引，可在 elasticsearch.yml设置：

action.destructive\_requires\_name: true

这个设置使删除只限于特定名称指向的数据, 而不允许通过指定 \_all 或通配符来删除指定索引库。

1. **索引设置**

在索引模板中将详解配置，两个最重要的设置：

* 1. number\_of\_shards：索引主分片数，默认值 5 ，该配置在索引创建后不能修改；
  2. number\_of\_replicas：主分片的副本数，默认值 1 ，对活动索引，该配置可修改。

PUT /my\_temp\_index

{

"settings": {

"number\_of\_shards" : 1,

"number\_of\_replicas" : 1

}

}

* 1. 配置分析器：默认情况下，停用词过滤器是被禁用的。

PUT /spanish\_docs{

"settings": {

"analysis": {

"analyzer": {

"es\_std": {

"type": "standard",

"stopwords": "\_spanish\_"

}

}}}}

1. **自定义分析器**

分析器包含三种函数，按照顺序被执行：

字符过滤器、分词器、词单元过滤器；

PUT /my\_index{

"settings": {

"analysis": {

"char\_filter": {

"&\_to\_and": {

"type": "mapping",

"mappings": [ "&=> and "]

}},

"filter": {

"my\_stopwords": {

"type": "stop",

"stopwords": [ "the", "a" ]

}},

"analyzer": {

"my\_analyzer": {

"**type**": "custom",

"char\_filter": [ "html\_strip", "&\_to\_and" ],

"tokenizer": "standard",

"filter": [ "lowercase", "my\_stopwords" ]

}}

}}}

1. **类型与映射**

类型可以很好的区分同一个集合中的不同细分。在不同的细分中数据的整体模式是相同的；

类型不适合完全不同类型的数据 。

1. **根对象**

映射的最高一层称为根对象，可能包含：

* 1. 一个properties节点，列出了文档中可能包含的每个字段的映射；
  2. 各种元数据字段，它们都以一个下划线开头，例如 \_type 、 \_id 和 \_source；
  3. 设置项，控制如何动态处理新的字段，如 analyzer 、 dynamic\_date\_formats 和dynamic\_templates；
  4. 其他设置，可以同时应用在根对象和其他 object 类型的字段上，如enabled、dynamic 和 include\_in\_all。

属性：type、index、analyzer

元数据\_source：一个储存字段，存储文档体的JSON字符串；

PUT /my\_index

{

"mappings": {

"my\_type": {

"\_source": {

"enabled": false

}

}

}

}

\_all 字段：把其它字段值 当作一个大字符串来索引的特殊字段。

PUT /my\_index/\_mapping/my\_type

{

"my\_type": {

"\_all": { "enabled": false }

}

}

 include\_in\_all ：设置逐个控制字段是否要包含在 \_all 字段中，默认值是 true；

1. **动态映射**

dynamic 配置来控制新字段是否加到文档，有三个值：

true: 动态添加新的字段--缺省；

false：忽略新的字段；

strict：如果遇到新字段抛出异常；

PUT /my\_index

{

"mappings": {

"my\_type": {

"dynamic": "strict",

"properties": {

"title": { "type": "string"},

"stash": {

"type": "object",

"dynamic": true

}

}

}

}

}

1. **自定义动态映射**

日期检测：

PUT /my\_index

{

"mappings": {

"my\_type": {

"date\_detection": false

}

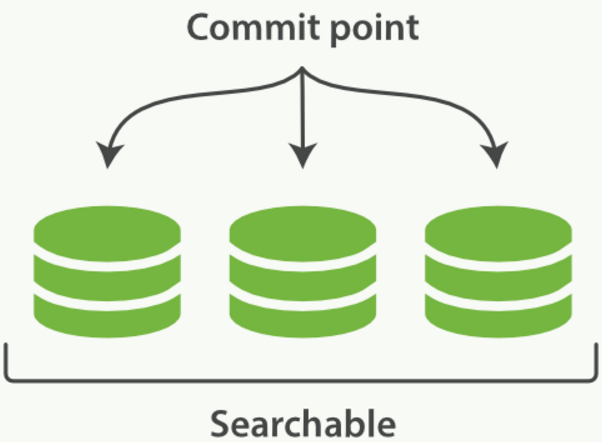
}

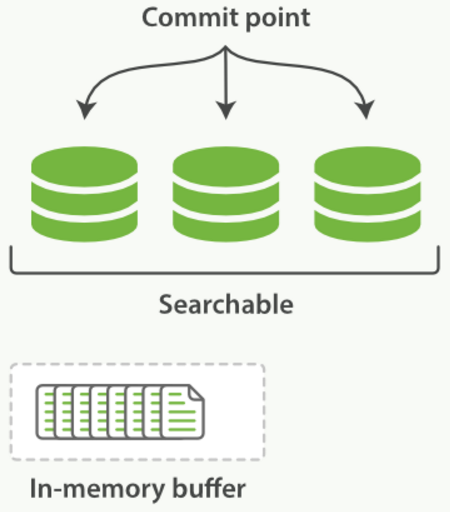
}

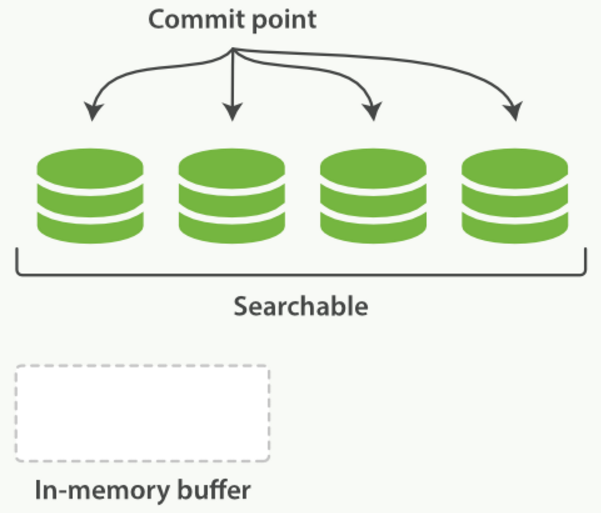
## 分片内部原理

1. **动态更新索引**

每个段本身都是一个倒排索引，而Lucene中的索引是很多段的集合和一个列出了所有已知段文件的提交点。







逐段搜索会以如下流程进行工作：

* + 1. 新文档被收集到内存索引缓存；
    2. 不时地, 缓存被提交 ：
       - 1. 一个新的段--一个追加的倒排索引--被写入磁盘；
         2. 一个新的包含新段名字的提交点被写入磁盘；
         3. 磁盘进行同步—所有在文件系统缓存中等待的写入都刷新到磁盘，以确保它们被写入物理文件；
    3. 新的段被开启，让它包含的文档可见以被搜索；
    4. 内存缓存被清空，等待接收新的文档。

1. **删除和更新**

段是不可改变的，触发删除操作后，每个提交点会包含一个.del的文件，文件中列出了被删除的文档的段信息。

当触发查询后，被删除的文档仍然可以被查询到，但是在结果返回之前从结果集合中移除。

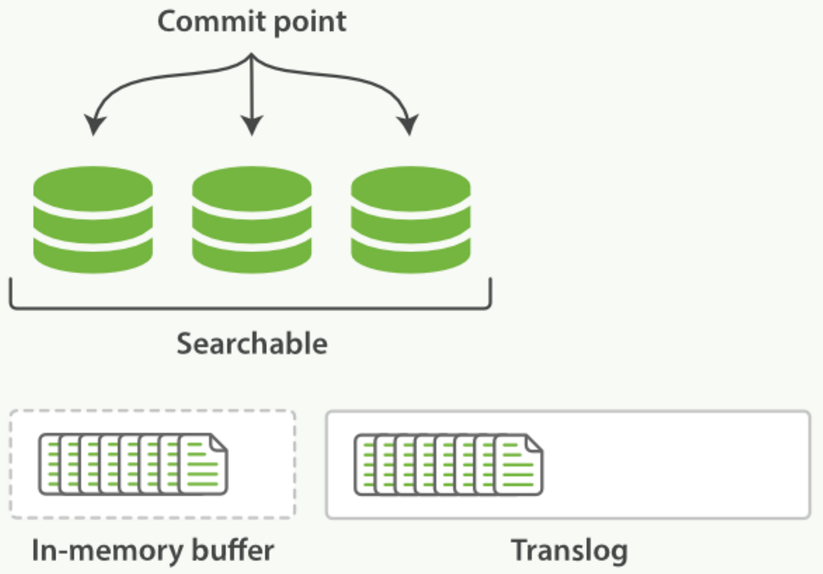
更新操作类似删除操作。

1. **近实时搜索和持久化**

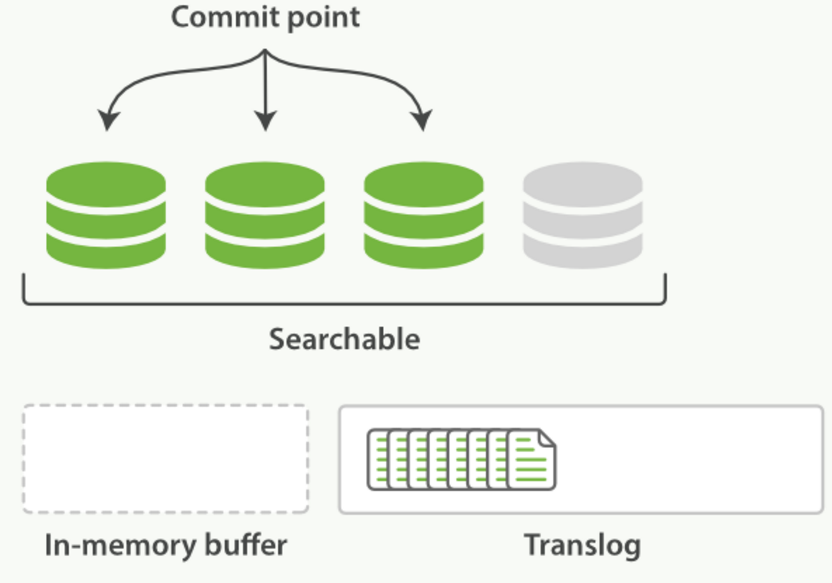
在Elasticsearch和磁盘之间是文件系统缓存。在内存索引缓冲区中的文档会被写入到一个新的段中。

从新文档索引写入磁盘大概分为几步：

1. 新文档索引写入内存缓冲区，并且追加到了事务日志Translog；

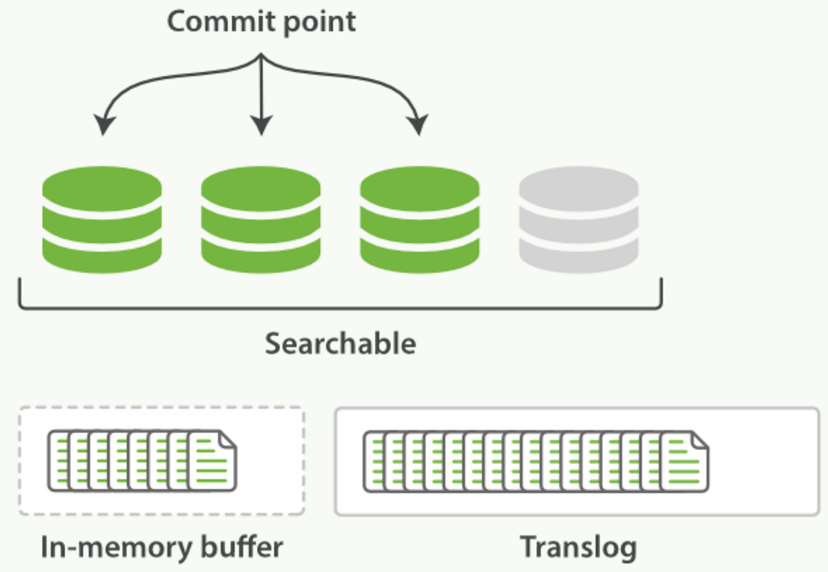


1. 内存中索引refresh到文件系统缓存中，索引可被搜索 --- 代价较低；

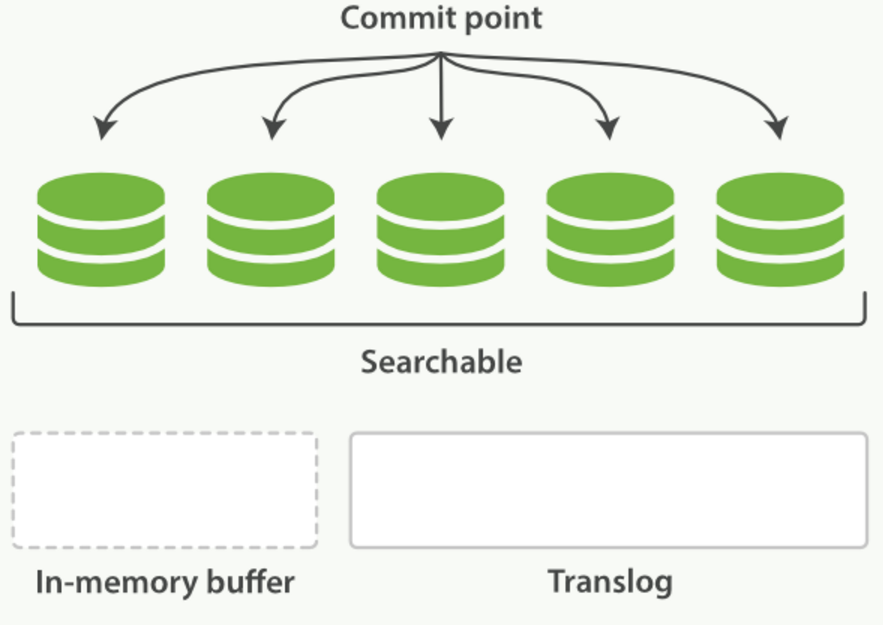


此时，可进行近实时搜索；

1. 进程继续工作，更多的文档被添加到内存缓冲区和追加到事务日志；



1. 每隔一段时间-- translog 变得越来越大--索引被刷新（flush）；一个新的 translog 被创建，并且段全量被提交； --- 代价较高。
   * + - 1. 所有在内存缓冲区的文档都被写入一个新的段；
         2. 缓冲区被清空；
         3. 一个提交点被写入硬盘；
         4. 文件系统缓存通过 fsync 被刷新（flush）；
         5. 老的 translog 被删除。



**refresh API：软提交**

在 Elasticsearch中，写入和打开一个新段的轻量的过程叫做 refresh；默认时间是1s，所以Elasticsearch默认的近实时搜索的实时性为1s。

POST /\_refresh 刷新所有索引

POST /blogs/\_refresh 只刷新blogs索引

PUT /my\_logs/\_settings 关闭自动刷新

{ "refresh\_interval": -1 }

**flush API： 硬提交**

执行一个提交并且截断 translog 的行为在 Elasticsearch 被称作一次 flush; 分片每30分钟被自动flush，或在 translog 达到512M（默认）也会flush。设置index.translog.flush\_threshold\_period和index.translog.flush\_threslog\_size参数。

POST /blogs/\_flush 刷新blogs索引

POST /\_flush?wait\_for\_ongoing 刷新所用索引，并等待操作结束返回

**建议：**在重启节点或关闭索引之前执行 [flush](https://www.elastic.co/guide/cn/elasticsearch/guide/current/translog.html#flush-api) 。当 Elasticsearch 尝试恢复或重新打开一个索引，它需要重放 translog中所有的操作，所以日志越短，恢复越快。

**Translog有多安全？**

translog目的是保证操作不丢失；默认translog每5秒被 fsync 刷新到硬盘。可以设置，但是性能有损失，特别是bulk操作。

Elasticsearch在CRUD请求操作在flush到主副分片的translog之后，才会向客户端返回200 OK的响应。这个同步操作保证了文档索引到内存缓存的安全。如果实际需求对丢失sync\_interval 时间段的数据无所谓，可以异步translog。

PUT /my\_index/\_settings{

"index.translog.durability": "async",

"index.translog.sync\_interval": "5s"

}

index.translog.flush\_threshold\_ops:发生多少次操作进行flush.默认是unlimited

index.translog.flush\_threshold\_size: translog大小达到此值时进行flush操作.默认512mb

index.translog.flush\_threshold\_period:在指定时间内进行一次强制flush操作.默认30m

index.translog.interval:多少时间间隔内会检查一次translog来进行一次flush操作.elasticsearch随机在这个值到这个值的2倍大小之间进行一次操作,默认是5s

index.gateway.local.sync:多少时间进行一次的写磁盘操作,默认是5s

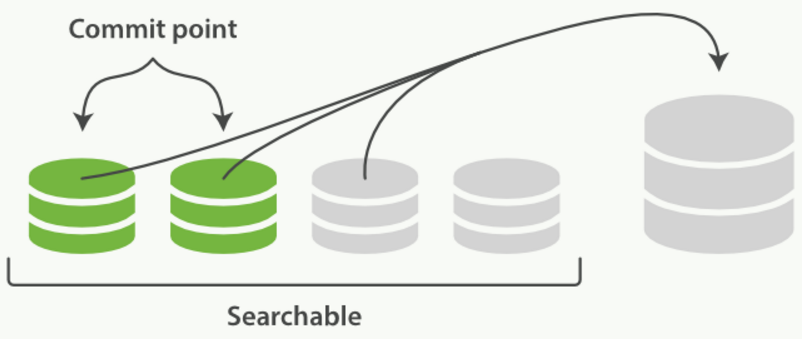
1. **段合并**

Elasticsearch自动刷新流程每秒创建一个段，短时间内端数据暴增。查询时遍历每个段的性能损耗较大，速度也越慢。

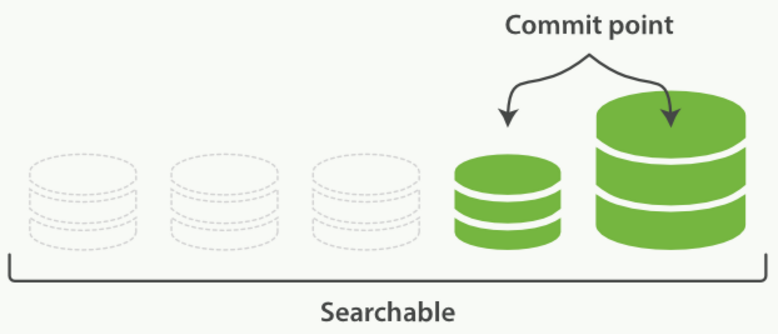
后台进行段合并可以解决该问题。段合并的时候会将那些旧的已删除文档 从文件系统中清除，被删除的文档不会被拷贝到新的大段中。段合并不影响索引和搜索。

段合并流程：

1. 当索引的时候，refresh操作会创建新段并将段打开以供搜索使用；
2. 合并进程选择小部分大小相似的段，并且在后台将它们合并到更大的段中。



1. 合并结束，老的段被删除
   * + - 1. 新的段和提交点被刷新到磁盘；
         2. 新的段被打开用来搜索；
         3. 老的段被删除。



optimize API

optimize API是强制合并API。它将一个分片强制合并到max\_num\_segments 参数指定大小的段数目，通常是减少到一个，提升搜索的性能。

POST /logstash-2014-10/\_optimize?max\_num\_segments=1

合并索引中的每个分片为一个单独的段。

index.merge.scheduler.max\_thread\_count: 1

注意：使用 optimize API 触发段合并的操作一点也不会受到任何资源上的限制。这可能会消耗掉节点上全部I/O资源, 使其没有余裕来处理搜索请求，从而有可能使集群失去响应。

**补充：**

段合并策略：tiered(默认)、log\_byte\_size、log\_doc；

# 深入搜索

1. **结构化搜索**
2. 精确值查找

当进行精确值查找时，请尽可能多的使用过滤式查询，不计算相关度且缓存。最常使用term查询，处理数字（numbers）、布尔值（Booleans）、日期（dates）以及文本（text）。用constant\_score结合filter查询。

内部过滤器的操作：

* + - * 1. 查询匹配文档；
        2. 创建bitset数组，包含0和1的数组，匹配的文档标志为1；在内部是一个Roaring bitmaps，稀疏或密集的集合进行高效编码。https://www.elastic.co/blog/frame-of-reference-and-roaring-bitmaps
        3. 迭代bitset，找到匹配的文档集合；
        4. 增量使用计数，Elasticsearch会为每个索引跟踪保留查询使用的历史状态。如果查询在最近的256次中查询中会被用到，那么就缓存到内存中。文档存在于低于总索引数3%的段不会被缓存。

1. 组合过滤器

结合bool和constant\_score；

1. 查找多个精确值

 term 和 terms 是 *包含（contains）* 操作，而非 *等值（equals）* （判断）。如果某个filed的值是数组，查找时包含即返回。

1. 范围

数值范围；

日期范围： range 查询支持对日期计算进行操作，在某个日期后加上一个双管符号 (||) 并紧跟一个日期数学表达式。

"lt" : "2014-01-01 00:00:00||+1M"

字符串范围：字典顺序，不建议使用。

1. 空值
2. 缓存
3. **全文搜索**

全文搜索两个最重要的方面：相关性和分析；

1. 基于词项查询和基于全文查询

基于词项查询：底层查询不需要分析阶段，它们对单个词项进行操作。

基于全文查询：

如果查询日期或整数，查询字符串分别作为日期和整数对待；

如果查询未分析的字段，整个查询字符串作为单个词项；

如果查询分析的字段，查询字符串传递到合适的分析器，然后，再查询。

1. 匹配查询

匹配查询 match 是个 核心 查询。无论查询什么字段， match 查询是首选查询方式。 它是一个高级全文查询 ，既能处理全文字段，又能处理精确字段。

match查询步骤：

检查字段类型 -> 分析查询字符串 -> 查找匹配文档 -> 为每个文档评分

1. 多词搜索与控制精度

"query": {

"match": {

"title": { https://www.elastic.co/guide/cn/elasticsearch/guide/current/images/icons/callouts/1.png

"query": "BROWN DOG!",

"operator": "and"

}

}

}

"minimum\_should\_match": "75%"

1. 组合查询与控制精度

"query": {

"bool": {

"should": [

{ "match": { "title": "brown" }},

{ "match": { "title": "fox" }},

{ "match": { "title": "dog" }}

],

"minimum\_should\_match": 2

}

}

1. **多字段搜索**
2. 多字符串查询

使用bool查询，boost改变评分。

1. 最佳字段

dis\_max 即分离 最大化查询；

"query": {

"dis\_max": {

"queries": [

{ "match": { "title": "Brown fox" }},

{ "match": { "body": "Brown fox" }}

]，

"tie\_breaker": 0.3

}

}

tie\_breaker 参数提供一种dis\_max和bool之间的折中选择：

* + - * 1. 获得最佳匹配语句的评分\_score;
        2. 将其它匹配语句的评分结果与tie\_breaker相乘；
        3. 对以上评分求和并规范化；

1. multi\_match查询

在多个字段上反复执行相同查询提供了一种便捷方式；

{

"multi\_match": {

"query": "Quick brown fox",

"type": "best\_fields",

"fields": [ "\*\_title", "body^2" ],

"tie\_breaker": 0.3,

"minimum\_should\_match": "30%"

}

}

{

"query": {

"multi\_match": {

"query": "Poland Street W1V",

"type": "most\_fields",

"fields": [ "street", "city", "country", "postcode" ]

}

}

}

1. **近似匹配**
2. 短语匹配

这里的短语匹配是指不仅匹配短语，位置也必须匹配；

"query": {

"match\_phrase": {

"title": "quick brown fox"

}

}

* + 1. quick 、 brown 和 fox 需要全部出现在域中；
    2. brown 的位置应该比 quick 的位置大1；
    3. fox 的位置应该比 quick 的位置大2。

1. 混合起来

slop 参数告诉 match\_phrase 查询词条相隔多远时仍然能将文档视为匹配 ；

"query": {

"match\_phrase": {

"title": {

"query": "quick fox",

"slop": 1

}}}

# 处理人类语言

在处理语言方面可优化的地方：归一化词元、将单词还原为词根、停用词：性能和精度、同义词、拼写错误和词汇识别等。

1. **使用语言分析器**

情景1：I'm not happy about the foxes 使用english分词器;

切词：i'm`，`happy`，`about`，`fox

如果查询not happy …，not是停用词被去掉了，查询时为了提高精确匹配文档的能力，使用[multifields](https://www.elastic.co/guide/cn/elasticsearch/guide/current/multi-fields.html)（多字段）对某字段建立两次索引。

"mappings": {

"blog": {

"properties": {

"title": {

"type": "string",

"fields": {

"english": {

"type": "string",

"analyzer": "english"

}

}

}

}

}

}

查询时使用most\_field多字段搜索语法：

"query": {

"multi\_match": {

"type": "most\_fields",

"query": "not happy foxes",

"fields": [ "title", "title.english" ]

}

}

1. **配置语言分析器**

情景1：no not不希望被听用掉；且Organization 不希望被词干成organ.

* 1. 词干提取排除
  2. 自定义停用词

PUT /my\_index{

"settings": {

"analysis": {

"analyzer": {

"my\_english": {

"type": "english",

"stem\_exclusion": [ "organization", "organizations" ],

"stopwords": [

"a", "an","with"…

]

}

}

}

}}

1. **混合语言的陷阱**

情景1：多语言处理

1) 每份doc有自己的主语言，包含一些其他语言片段；

2) 每个field有自己的主语言，包含一些其他语言片段；

3) 每个field都是混合语言；

1) 每份文档一种语言

不同语言的文档被分别存放在不同的索引中 — blogs-en 、blogs-fr；

PUT /blogs-en{

"mappings": {

"post": {

"properties": {

"title": {

"type": "string",

"fields": {

"stemmed": {

"type": "string",

"analyzer": "english"

}}}}}}}

PUT /blogs-fr{

"mappings": {

"post": {

"properties": {

"title": {

"type": "string",

"fields": {

"stemmed": {

"type": "string",

"analyzer": "french"

}}}}}}}

indices\_boost 参数为特定的语言添加优先权；

"query":{

"multi\_match": {

"query": "deja vu",

"fields": [ "title", "title.stemmed" ]

"type": "most\_fields"

}

},

"indices\_boost": {

"blogs-en": 3,

"blogs-fr": 2

}

2) 每个域一种语言

"mappings": {

"movie": {

"properties": {

"title": {

"type": "string"

},

"title\_br": {

"type": "string",

"analyzer": "brazilian"

},

"title\_en": {

"type": "string",

"analyzer": "english"

}}}}

"query": {

"multi\_match": {

"query": "club de la lucha",

"fields": [ "title\*", "title\_en^2" ],

"type": "most\_fields"

}}

3) 混合语言 略

1. **词汇识别**
   * 1. 标准分析器

任何全文检索的字符串域都默认使用 standard 分析器。可自定义重新实现标准分析器。

* + 1. 标准分词器

**分词器：**接受一个字符串作为输入，将这个字符串拆分成独立的词或语汇单元，可能会丢弃一些标点符号等，输出一个语汇单元流。

standard分词器使用Unicode文本分割算法来寻找单次之间的界限。

* + 1. 安装ICU插件

Elasticsearch的 [ICU分析器插件](https://github.com/elasticsearch/elasticsearch-analysis-icu)使用国际化组件Unicode (ICU)函数库提供丰富的处理Unicode工具。

./bin/plugin -install elasticsearch/elasticsearch-analysis-icu/$VERSION

https://github.com/elastic/elasticsearch-analysis-icu

* + 1. icu\_分词器

icu\_分词器 和 标准分词器 使用同样的Unicode文本分段算法， 只是为了更好的支持亚洲语。

* + 1. 整理输入文本

添加字符过滤器到分析器中，对文本在传给分词器之前预处理该文本。比如HTML分词，去除HTML标签；比如处理标点符号。

1. **归一化词元**

**语汇单元过滤器**的工作主要负责归一化词元的工作。它接收来自分词器(tokenizer)的词元(token)流。

常用的语汇单元过滤器是lowercase 过滤器；asciifolding 字符过滤器来实现去掉变音符号。

"settings": {

"analysis": {

"analyzer": {

"folding": {

"tokenizer": "standard",

"filter": [ "lowercase", "asciifolding" ]

}}}}

1. **将单词还原为词根**

**词干提取**：试图移除单词的变化形式之间的差别，从而达到将每个词都提取为它的词根形式。

词干提取是一种遭受两种困扰的模糊的技术：**词干弱提取**和**词干过度提取。**

1. **词干提取算法**

基于算法的 stemmers 优点是可以作为插件使用，速度快，占用内存少，有规律的单词处理效果好。

"settings": {

"analysis": {

"filter": {

"english\_stop": {

"type": "stop",

"stopwords": "\_english\_"

},

"light\_english\_stemmer": {

"type": "stemmer",

"language": "light\_english" //非激进的 [kstem](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-kstem-tokenfilter.html) 分词过滤器的映射

},

"english\_possessive\_stemmer": {

"type": "stemmer",

"language": "possessive\_english"

} },

"analyzer": {

"english": {

"tokenizer": "standard",

"filter": [

"english\_possessive\_stemmer",

"lowercase",

"english\_stop",

"light\_english\_stemmer",

"asciifolding"

]

}}}}

1. **字典词干提取器**

实践中一个好的算法化词干提取器一般优于一个字典词干提取器。主要有两大原因：字典质量和大小与性能。

1. **Hunspell词干提取器**
2. **词干提取器**
   1. English [porter\_stem](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-porterstem-tokenfilter.html) 语汇单元过滤器（token filter）
   2. light\_english [kstem](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-kstem-tokenfilter.html) 语汇单元过滤器（token filter）
   3. minimal\_english Lucene的 EnglishMinimalStemmer，移除复数
   4. lovins 基于 [Snowball](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-snowball-tokenfilter.html)的[Lovins](http://snowball.tartarus.org/algorithms/lovins/stemmer.html) 提取器
   5. porter 基于 [Snowball](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-snowball-tokenfilter.html) 的 [Porter](http://snowball.tartarus.org/algorithms/porter/stemmer.html) 提取器
   6. porter2 基于 [Snowball](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-snowball-tokenfilter.html) 的 [Porter2](http://snowball.tartarus.org/algorithms/english/stemmer.html) 提取器
   7. possessive\_english Lucene里面的EnglishPossessiveFilter ，移除 's

选择一个词干提取器，主要考虑三个方面的因素：提取性能，提取质量和提取程度。

1. **控制词干提取**

语汇单元过滤器 [keyword\_marker](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-keyword-marker-tokenfilter.html) 和 [stemmer\_override](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-stemmer-override-tokenfilter.html) 能让我们自定义词干提取过程。

语言分析器的参数 [stem\_exclusion](https://www.elastic.co/guide/cn/elasticsearch/guide/current/configuring-language-analyzers.html#stem-exclusion) 允许我们指定一个词语列表，让他们不被词干提取。语言分析器使用 [keyword\_marker 语汇单元过滤器](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-keyword-marker-tokenfilter.html) 来标记这些词语列表为 *keywords* ，用来阻止后续的词干提取过滤器来触碰这些词语。

自定义提取：[stemmer\_override](https://www.elastic.co/guide/en/elasticsearch/reference/master/analysis-stemmer-override-tokenfilter.html) 语汇单元过滤器允许我们指定自定义的提取规则。

1. **原形词干提取**

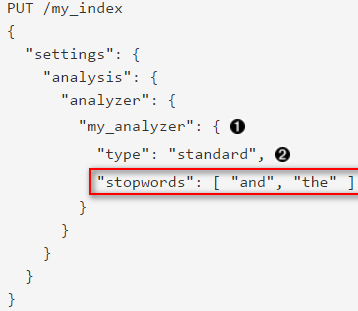
基于这些原因，我们不推荐使用原形词干提取。

1. **停用词**

低频词和高频词；

停用词的引进可以节省存储空间，但是降低了搜索能力。

1. **使用停用词**

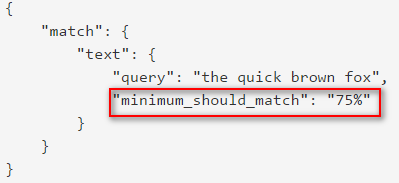


也可以通过"stopwords": "\_english\_"指定停用词

1. **停用词与性能**
   1. **and操作符**

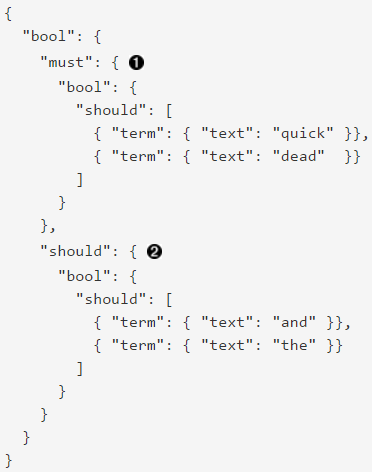
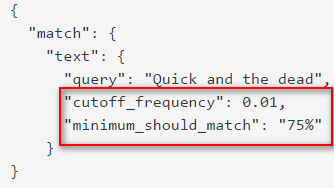
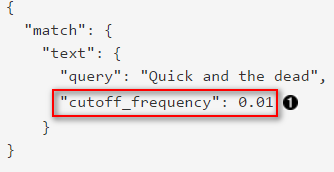
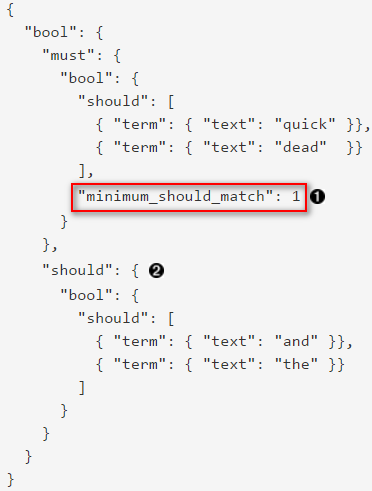


* 1. **最少匹配数**



1. **词项的分别管理**

match 查询接受一个参数 cutoff\_frequency ，从而可以让它将查询字符串里的词项分为低频和高频两组。

1. **停用词与短语查询**

一个典型的索引会可能包含部分或所有以下数据：

词项字典：索引中所有文档内所有词项有序列表，及包含该词的文档数量；

倒排表：包含每个词项的文档（ID）列表；

词频：每个词项在文档里出现的频率；

位置：每个词项在文档里出现的位置，供短语查询或近似查询使用；

偏移：每个词项在文档开始与结束字符偏移，供高亮使用，默认禁用；

规范因子：用对字段长度进行规范化处理的因子，给较短字段予以更多权重。

**索引选项**

情景1：是否真的需要使用短语查询或近似查询？（不需要）

index\_options 参数 控制索引里为每个字段存储的信息， 可选值如下:

docs：只存储文档及其包含词项的信息， not\_analyzed 字符串字段是默认的；

freqs：存储 docs 信息，以及每个词在每个文档里出现的频次；

positions：存储 docs , freqs ,analyzed ，以及每个词项在文档里出现的位置；

offsets：存储 docs,freqs,positions, 以及每个词在原始字符串中开始与结束字符的偏移信息；这个信息被用以高亮搜索结果，它默认是禁用的。



# 聚合

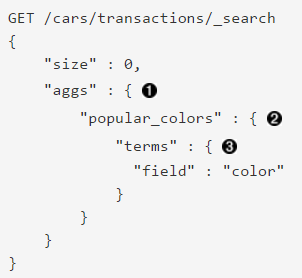
* 1. **高阶概念**

桶(*Buckets*)： 满足特定条件的文档集合, 基于条件来划分文档；--- GROUP BY color

指标(*Metrics*)：对桶内的文档进行统计计算, 大多数 *指标* 是简单的数学运算（例最小值、平均值、最大值，还有汇总）；--- COUNT(color)

**聚合** 是由桶和指标组成的。

例1：统计每种颜色额车总数；每个颜色动态创建新桶；



例2：在桶上加上度量指标；计算每个颜色桶里的价格平均值；



例3：嵌套桶； 在颜色桶内嵌套一个make的桶；得到组合结果(color, make);



例4：在制造商桶内添加度量指标，最大最小值；



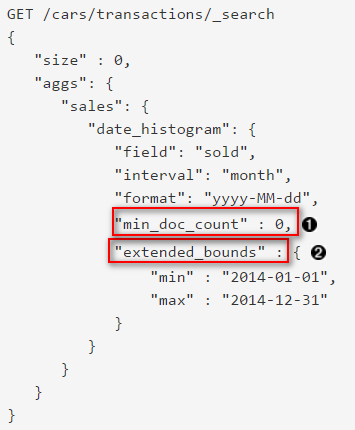
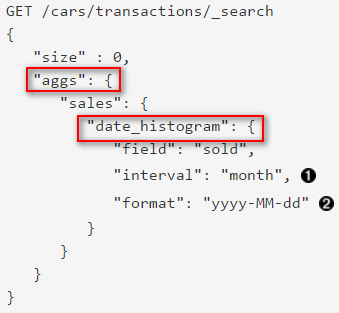
例5：条形图；每个售价区间内汽车的销量；



histogram 桶要求两个参数：一个数值字段以及一个定义桶大小间隔。

* 1. **按时间统计**

在时间维度上构建指标：date\_histogram 可以自动识别月份天数，处理时区等。







* 1. **范围限定的聚合**

聚合是基于我们查询匹配的文档集合进行计算的；



全局桶：包含所有的文档，它无视查询的范围。



* 1. **过滤和聚合**

聚合是在查询结果范围内操作的，任何可以适用于查询的过滤器也可以应用在聚合上。

**过滤：**



**过滤桶：**对聚合结果过滤；

****

**后过滤器**：只过滤搜索结果，不过滤聚合结果；



* 1. **多桶排序**

多值桶（ terms 、 histogram 和 date\_histogram ）动态生成很多桶。 Elasticsearch 是如何决定这些桶展示给用户的顺序呢？

**内置排序：**

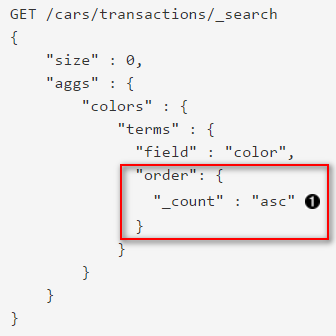
为聚合引入了一个 order 对象， 它可以根据以下几个值中的一个值进行排序：

\_count：按文档数排序。对 terms 、 histogram 、 date\_histogram 有效；

\_term：按词项的字符串值的字母顺序排序。只在 terms 内使用；

\_key：按每个桶的键值数值排序（理论上与 \_term 类似）；

只在 histogram 和 date\_histogram 内使用。



**度量排序：**

度量指标中指定 order 参数即可；



使用多值度量进行排序，使用点式路径；



* 1. **基于“深度”度量排序**

对更深 的度量进行排序，用尖括号（>）嵌套起来：

my\_bucket>another\_bucket>metric

需要提醒的是嵌套路径上的每个桶都必须是单值的。

单值桶： filter 、global 和 reverse\_nested；



# 地理位置

# 数据建模

# 管理、监控和部署

# 附录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **序号** | **变更时间** | **版本** | **变更人** | **变更说明** |
| 1 | 2017.05.03 | 1.0 | 李聪8 | 新建《Elasticsearch基础入门》 |