# 什么是Elasticsearch

Elasticsearch，分布式，高性能，高可用，可伸缩的搜索和分析系统

## 什么是Elasticsearch

### 什么是搜索？

百度：我们比如说想找寻任何的信息的时候，就会上百度去搜索一下，比如说找一部自己喜欢的电影，或者说找一本喜欢的书，或者找一条感兴趣的新闻（提到搜索的第一印象）

百度 != 搜索，这是不对的

垂直搜索（站内搜索）

互联网的搜索：电商网站，招聘网站，新闻网站，各种app

IT系统的搜索：OA软件，办公自动化软件，会议管理，日程管理，项目管理，员工管理，搜索“张三”，“张三儿”，“张小三”；有个电商网站，卖家，后台管理系统，搜索“牙膏”，订单，“牙膏相关的订单”

搜索，就是在任何场景下，找寻你想要的信息，这个时候，会输入一段你要搜索的关键字，然后就期望找到这个关键字相关的有些信息

### 如果用数据库做搜索会怎么样？

做软件开发的话，或者对IT、计算机有一定的了解的话，都知道，数据都是存储在数据库里面的，比如说电商网站的商品信息，招聘网站的职位信息，新闻网站的新闻信息，等等吧。所以说，很自然的一点，如果说从技术的角度去考虑，如何实现如说，电商网站内部的搜索功能的话，就可以考虑，去使用数据库去进行搜索。

1、 比方说，每条记录的指定字段的文本，可能会很长，比如说“商品描述”字段的长度，有长达数千个，甚至数万个字符，这个时候，每次都要对每条记录的所有文本进行扫描，懒判断说，你包不包含我指定的这个关键词（比如说“牙膏”）

2、 还不能将搜索词拆分开来，尽可能去搜索更多的符合你的期望的结果，比如输入“生化

机”，就搜索不出来“生化危机”

用数据库来实现搜索，是不太靠谱的。通常来说，性能会很差的。

### 什么是全文检索和Lucene？

1. 全文检索，倒排索引
2. lucene，就是一个jar包，里面包含了封装好的各种建立倒排索引，以及进行搜

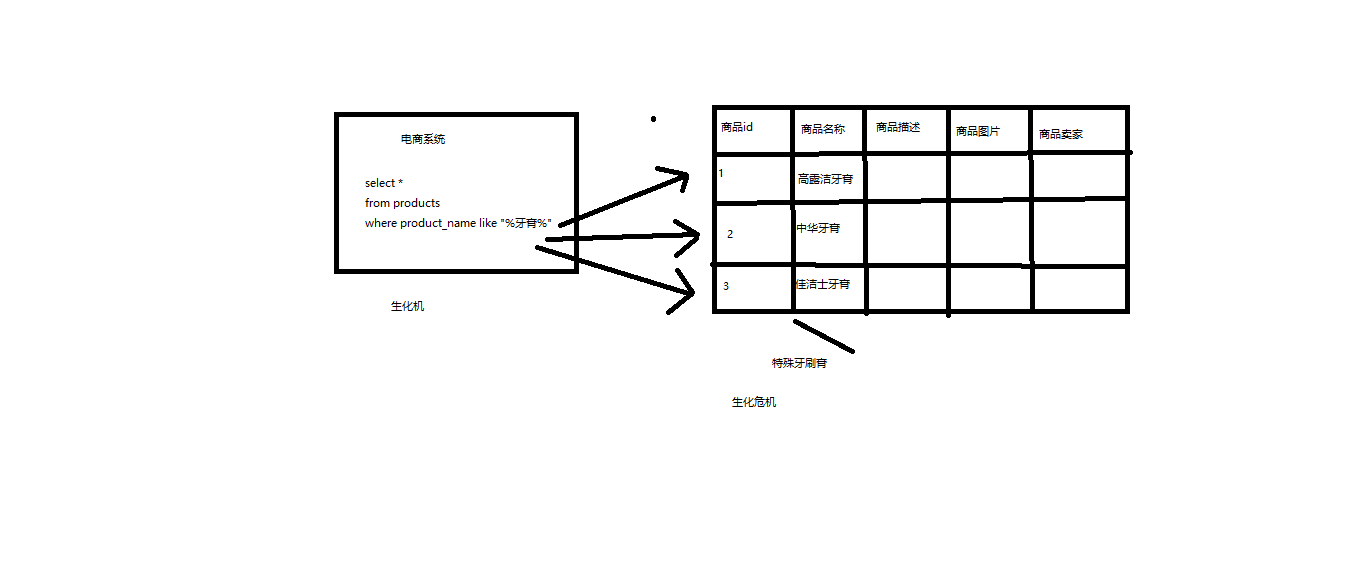
索的代码，包括各种算法。我们就用java开发的时候，引入lucene jar，然后

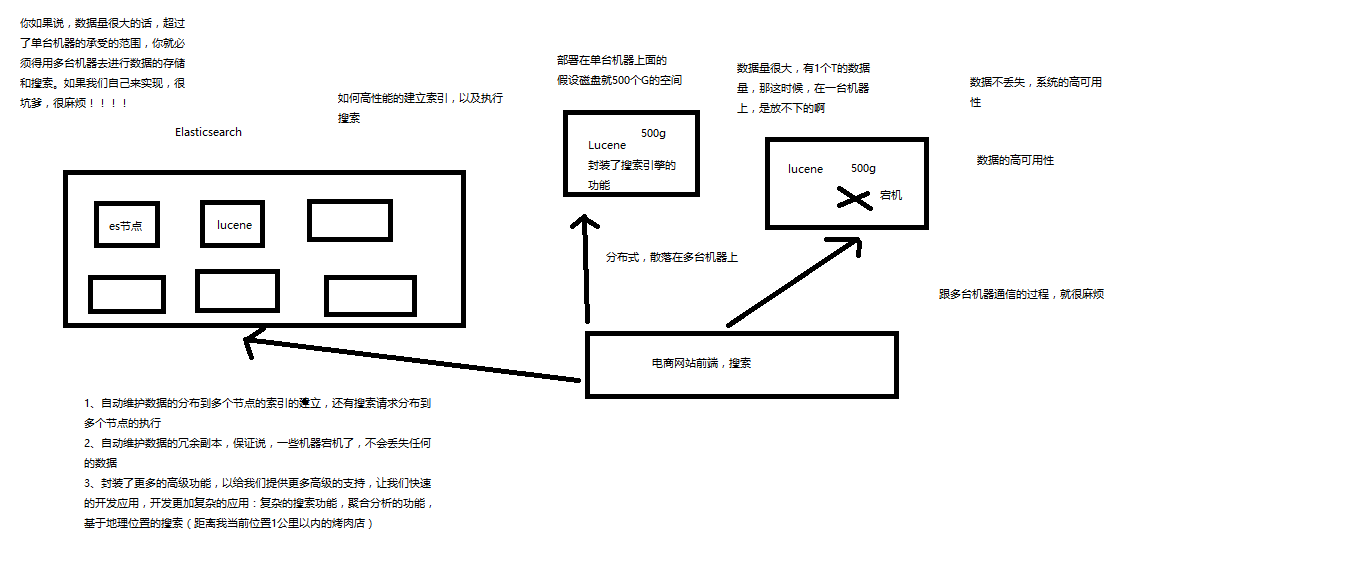
基于lucene的api进行去进行开发就可以了。用lucene，我们就可以去将已

有的数据建立索引，lucene会在本地磁盘上面，给我们组织索引的数据结构。另

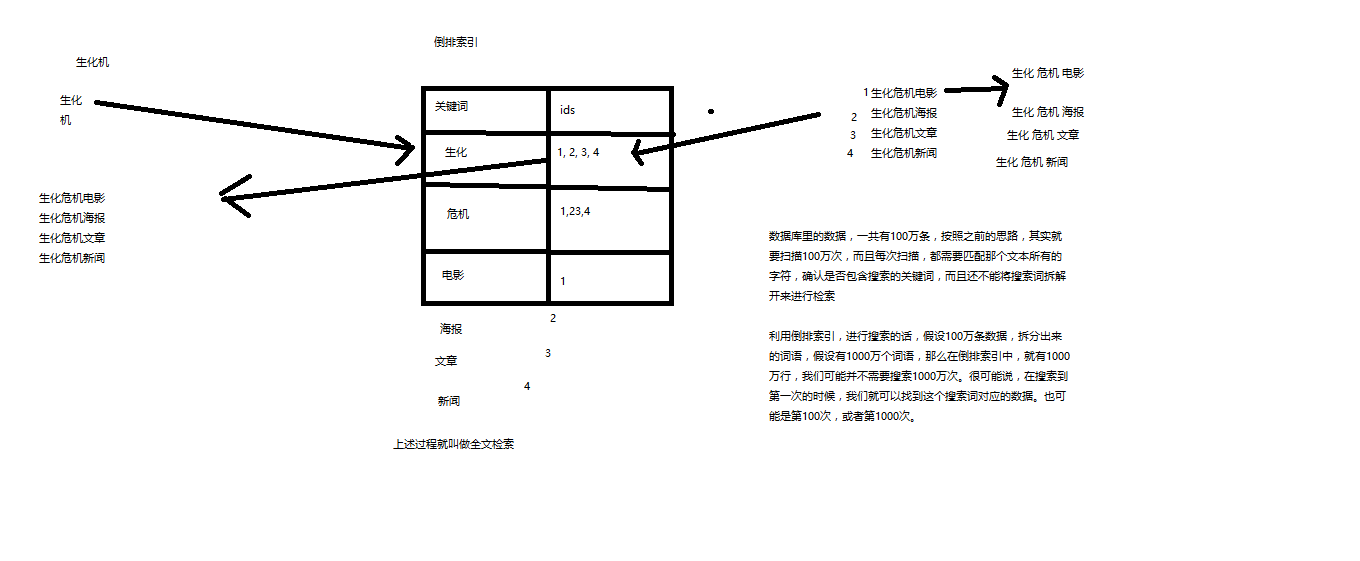
外的话，我们也可以用lucene提供的一些功能和api来针对磁盘上额

### 什么是Elasticsearch？

（1）图解分析

**如果用数据库做搜索会怎么样**

**什么是Elasticsearch**



**什么是全文检索**

## 功能、适用场景以及特点介绍

### 分布式的搜索引擎和数据分析引擎

搜索：百度，网站的站内搜索，IT系统的检索

数据分析：电商网站，最近7天牙膏这种商品销量排名前10的商家有哪些；新闻网站，最近1个月访问量排名前3的新闻版块是哪些

分布式，搜索，数据分析

### 全文检索，结构化检索，数据分析

全文检索：我想搜索商品名称包含牙膏的商品，select \* from products where product\_name like "%牙膏%"

结构化检索：我想搜索商品分类为日化用品的商品都有哪些，select \* from products where category\_id='日化用品'

部分匹配、自动完成、搜索纠错、搜索推荐

数据分析：我们分析每一个商品分类下有多少个商品，select category\_id,count(\*) from products group by category\_id

### 对海量数据进行近实时的处理

分布式：ES自动可以将海量数据分散到多台服务器上去存储和检索

海联数据的处理：分布式以后，就可以采用大量的服务器去存储和检索数据，自然而然就可以实现海量数据的处理了

近实时：检索个数据要花费1小时（这就不要近实时，离线批处理，batch-processing）；在秒级别对数据进行搜索和分析

跟分布式/海量数据相反的：lucene，单机应用，只能在单台服务器上使用，最多只能处理单台服务器可以处理的数据量

### Elasticsearch的适用场景

国外

* 维基百科，类似百度百科，牙膏，牙膏的维基百科，全文检索，高亮，搜索推荐
* The Guardian（国外新闻网站），类似搜狐新闻，用户行为日志（点击，浏览，收藏，评论）+社交网络数据（对某某新闻的相关看法），数据分析，给到每篇新闻文章的作者，让他知道他的文章的公众反馈（好，坏，热门，垃圾，鄙视，崇拜）
* Stack Overflow（国外的程序异常讨论论坛），IT问题，程序的报错，提交上去，有人会跟你讨论和回答，全文检索，搜索相关问题和答案，程序报错了，就会将报错信息粘贴到里面去，搜索有没有对应的答案
* GitHub（开源代码管理），搜索上千亿行代码
* 电商网站，检索商品
* 日志数据分析，logstash采集日志，ES进行复杂的数据分析（ELK技术，elasticsearch+logstash+kibana）
* 商品价格监控网站，用户设定某商品的价格阈值，当低于该阈值的时候，发送通知消息给用户，比如说订阅牙膏的监控，如果高露洁牙膏的家庭套装低于50块钱，就通知我，我就去买
* BI系统，商业智能，Business Intelligence。比如说有个大型商场集团，BI，分析一下某某区域最近3年的用户消费金额的趋势以及用户群体的组成构成，产出相关的数张报表，\*\*区，最近3年，每年消费金额呈现100%的增长，而且用户群体85%是高级白领，开一个新商场。ES执行数据分析和挖掘，Kibana进行数据可视化

国内

* 国内：站内搜索（电商，招聘，门户，等等），IT系统搜索（OA，CRM，ERP

等等），数据分析（ES热门的一个使用场景）

### Elasticsearch的特点

1. 可以作为一个大型分布式集群（数百台服务器）技术，处理PB级数据，服务大公司；也可以运行在单机上，服务小公司
2. Elasticsearch不是什么新技术，主要是将全文检索、数据分析以及分布式技术，合并在了一起，才形成了独一无二的ES；lucene（全文检索），商用的数据分析软件（也是有的），分布式数据库（mycat）
3. 对用户而言，是开箱即用的，非常简单，作为中小型的应用，直接3分钟部署一下ES，就可以作为生产环境的系统来使用了，数据量不大，操作不是太复杂
4. 数据库的功能面对很多领域是不够用的（事务，还有各种联机事务型的操作）；特殊的功能，比如全文检索，同义词处理，相关度排名，复杂数据分析，海量数据的近实时处理；Elasticsearch作为传统数据库的一个补充，提供了数据库所不不能提供的很多功能

## NRT、索引、分片、副本等概念

### lucene和elasticsearch的前世今生

lucene，最先进、功能最强大的搜索库，直接基于lucene开发，非常复杂，api复杂（实现一些简单的功能，写大量的java代码），需要深入理解原理（各种索引结构）

elasticsearch，基于lucene，隐藏复杂性，提供简单易用的restful api接口、java api接口（还有其他语言的api接口）

1. 分布式的文档存储引擎
2. 分布式的搜索引擎和分析引擎
3. 分布式，支持PB级数据

开箱即用，优秀的默认参数，不需要任何额外设置，完全开源

关于elasticsearch的一个传说，有一个程序员失业了，陪着自己老婆去英国伦敦学习厨师课程。程序员在失业期间想给老婆写一个菜谱搜索引擎，觉得lucene实在太复杂了，就开发了一个封装了lucene的开源项目，compass。后来程序员找到了工作，是做分布式的高性能项目的，觉得compass不够，就写了elasticsearch，让lucene变成分布式的系统。

### elasticsearch的核心概念

1. Near Realtime（NRT）：近实时，两个意思，从写入数据到数据可以被搜索到有一个小延迟（大概1秒）；基于es执行搜索和分析可以达到秒级
2. Cluster：集群，包含多个节点，每个节点属于哪个集群是通过一个配置（集群名称，默认是elasticsearch）来决定的，对于中小型应用来说，刚开始一个集群就一个节点很正常
3. Node：节点，集群中的一个节点，节点也有一个名称（默认是随机分配的），节点名称很重要（在执行运维管理操作的时候），默认节点会去加入一个名称为“elasticsearch”的集群，如果直接启动一堆节点，那么它们会自动组成一个elasticsearch集群，当然一个节点也可以组成一个elasticsearch集群
4. Document&field：文档，es中的最小数据单元，一个document可以是一条客户数据，一条商品分类数据，一条订单数据，通常用JSON数据结构表示，每个index下的type中，都可以去存储多个document。一个document里面有多个field，每个field就是一个数据字段。

product document

{

"product\_id": "1",

"product\_name": "高露洁牙膏",

"product\_desc": "高效美白",

"category\_id": "2",

"category\_name": "日化用品"

}

1. Index：索引，包含一堆有相似结构的文档数据，比如可以有一个客户索引，商品分类索引，订单索引，索引有一个名称。一个index包含很多document，一个index就代表了一类类似的或者相同的document。比如说建立一个product index，商品索引，里面可能就存放了所有的商品数据，所有的商品document。
2. Type：类型，每个索引里都可以有一个或多个type，type是index中的一个逻辑数据分类，一个type下的document，都有相同的field，比如博客系统，有一个索引，可以定义用户数据type，博客数据type，评论数据type。

商品index，里面存放了所有的商品数据，商品document

但是商品分很多种类，每个种类的document的field可能不太一样，比如说电器商品，可能还包含一些诸如售后时间范围这样的特殊field；生鲜商品，还包含一些诸如生鲜保质期之类的特殊field

type，日化商品type，电器商品type，生鲜商品type

日化商品type：product\_id，product\_name，product\_desc，category\_id，category\_name

电器商品type：product\_id，product\_name，product\_desc，category\_id，category\_name，service\_period

生鲜商品type：product\_id，product\_name，product\_desc，category\_id，category\_name，eat\_period

每一个type里面，都会包含一堆document

{

"product\_id": "2",

"product\_name": "长虹电视机",

"product\_desc": "4k高清",

"category\_id": "3",

"category\_name": "电器",

"service\_period": "1年"

}

{

"product\_id": "3",

"product\_name": "基围虾",

"product\_desc": "纯天然，冰岛产",

"category\_id": "4",

"category\_name": "生鲜",

"eat\_period": "7天"

}

1. shard：单台机器无法存储大量数据，es可以将一个索引中的数据切分为多个shard，分布在多台服务器上存储。有了shard就可以横向扩展，存储更多数据，让搜索和分析等操作分布到多台服务器上去执行，提升吞吐量和性能。每个shard都是一个lucene index。
2. replica：任何一个服务器随时可能故障或宕机，此时shard可能就会丢失，因此可以为每个shard创建多个replica副本。replica可以在shard故障时提供备用服务，保证数据不丢失，多个replica还可以提升搜索操作的吞吐量和性能。primary shard（建立索引时一次设置，不能修改，默认5个），replica shard（随时修改数量，默认1个），默认每个索引10个shard，5个primary shard，5个replica shard，最小的高可用配置，是2台服务器。

### elasticsearch核心概念 vs. 数据库核心概念

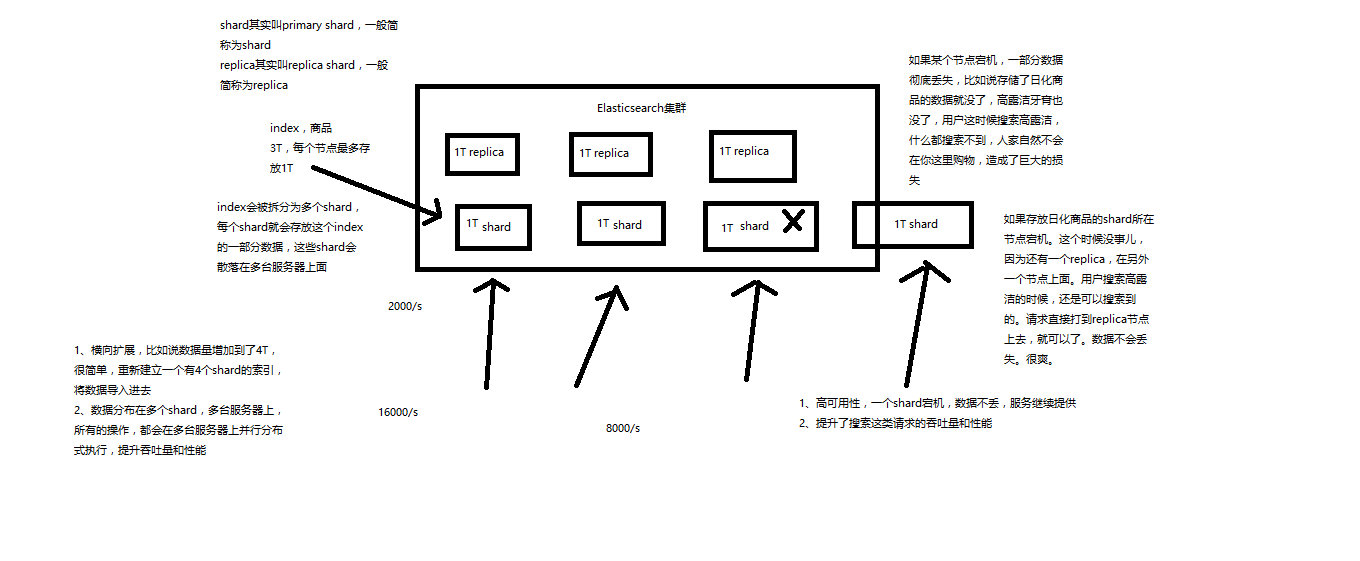
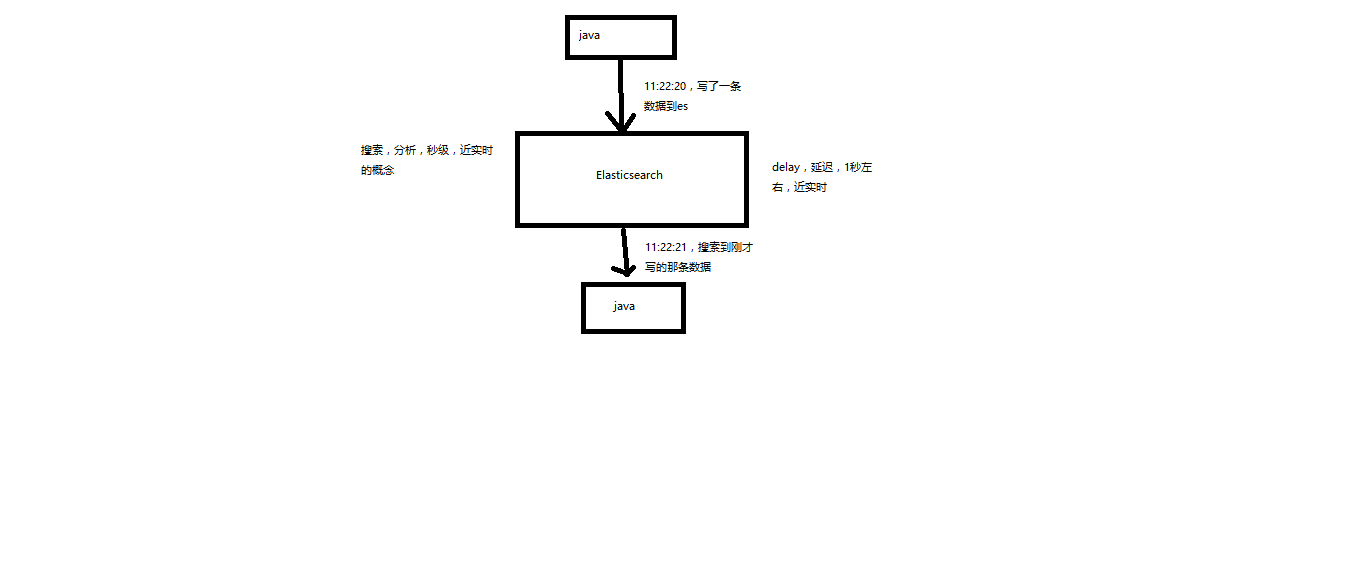
Elasticsearch 数据库

---------------------------------------------------------

Document 行

Type 表

Index 库

**Elasticsearch近实时概念的解释**

**shard和replica的解释**

## windows上安装和启动Elasticseach

1. 安装JDK，至少1.8.0\_73以上版本，java –version
2. 下载和解压缩Elasticsearch安装包，目录结构
3. 启动Elasticsearch：bin\elasticsearch.bat，es本身特点之一就

是开箱即用，如果是中小型应用，数据量少，操作不是很复杂，直接启动就可以用了

1. 检查ES是否启动成功：http://localhost:9200/?pretty

name: node名称

cluster\_name: 集群名称（默认的集群名称就是elasticsearch）

version.number: 5.2.0，es版本号

{

"name" : "4onsTYV",

"cluster\_name" : "elasticsearch",

"cluster\_uuid" : "nKZ9VK\_vQdSQ1J0Dx9gx1Q",

"version" : {

"number" : "5.2.0",

"build\_hash" : "24e05b9",

"build\_date" : "2017-01-24T19:52:35.800Z",

"build\_snapshot" : false,

"lucene\_version" : "6.4.0"

},

"tagline" : "You Know, for Search"

}

1. 修改集群名称：elasticsearch.yml
2. 下载和解压缩Kibana安装包，使用里面的开发界面，去操作

elasticsearch,作为我们学习es知识点的一个主要的界面入口

1. 启动Kibana：bin\kibana.bat
2. 进入Dev Tools界面
3. GET \_cluster/health

## 集群健康检查，文档CRUD

* + 1. document数据格式

面向文档的搜索分析引擎

1. 应用系统的数据结构都是面向对象的，复杂的
2. 对象数据存储到数据库中，只能拆解开来，变为扁平的多张表，每次查询的时 候还得还原回对象格式，相当麻烦
3. ES是面向文档的，文档中存储的数据结构，与面向对象的数据结构是一样的，基于这种文档数据结构，es可以提供复杂的索引，全文检索，分析聚合等功能
4. es的document用json数据格式来表达

public class Employee {

private String email;

private String firstName;

private String lastName;

private EmployeeInfo info;

private Date joinDate;

}

private class EmployeeInfo {

private String bio; // 性格

private Integer age;

private String[] interests; // 兴趣爱好

}

EmployeeInfo info = new EmployeeInfo();

info.setBio("curious and modest");

info.setAge(30);

info.setInterests(new String[]{"bike", "climb"});

Employee employee = new Employee();

employee.setEmail("zhangsan@sina.com");

employee.setFirstName("san");

employee.setLastName("zhang");

employee.setInfo(info);

employee.setJoinDate(new Date());

employee对象：里面包含了Employee类自己的属性，还有一个EmployeeInfo对象

两张表：employee表，employee\_info表，将employee对象的数据重新拆开来，变成Employee数据和EmployeeInfo数据

employee表：email，first\_name，last\_name，join\_date，4个字段

employee\_info表：bio，age，interests，3个字段；此外还有一个外键字段，比如employee\_id，关联着employee表

{

"email": "zhangsan@sina.com",

"first\_name": "san",

"last\_name": "zhang",

"info": {

"bio": "curious and modest",

"age": 30,

"interests": [ "bike", "climb" ]

},

"join\_date": "2017/01/01"

}

我们就明白了es的document数据格式和数据库的关系型数据格式的区别

* + 1. 电商网站商品管理案例背景介绍

有一个电商网站，需要为其基于ES构建一个后台系统，提供以下功能：

（1）对商品信息进行CRUD（增删改查）操作

（2）执行简单的结构化查询

（3）可以执行简单的全文检索，以及复杂的phrase（短语）检索

（4）对于全文检索的结果，可以进行高亮显示

（5）对数据进行简单的聚合分析

* + 1. 简单的集群管理

### 快速检查集群的健康状况

es提供了一套api，叫做cat api，可以查看es中各种各样的数据

GET /\_cat/health?v

C:\Users\SMILES~1\AppData\Local\Temp\1562395636(1).png

与上图类似

epoch timestamp cluster status node.total node.data shards pri relo init unassign pending\_tasks max\_task\_wait\_time active\_shards\_percent

1488006741 15:12:21 elasticsearch yellow 1 1 1 1 0 0 1 0 - 50.0%

epoch timestamp cluster status node.total node.data shards pri relo init unassign pending\_tasks max\_task\_wait\_time active\_shards\_percent

1488007113 15:18:33 elasticsearch green 2 2 2 1 0 0 0 0 - 100.0%

如何快速了解集群的健康状况？green、yellow、red？

green：每个索引的primary shard和replica shard都是active状态的

yellow：每个索引的primary shard都是active状态的，但是部分replica shard不是active状态，处于不可用的状态

red：不是所有索引的primary shard都是active状态的，部分索引有数据丢失了

为什么现在会处于一个yellow状态？

我们现在就一个笔记本电脑，就启动了一个es进程，相当于就只有一个node。现在es中有一个index，就是kibana自己内置建立的index。由于默认的配置是给每个index分配5个primary shard和5个replica shard，而且primary shard和replica shard不能在同一台机器上（为了容错）。现在kibana自己建立的index是1个primary shard和1个replica shard。当前就一个node，所以只有1个primary shard被分配了和启动了，但是一个replica shard没有第二台机器去启动。

做一个小实验：此时只要启动第二个es进程，就会在es集群中有2个node，然后那1个replica shard就会自动分配过去，然后cluster status就会变成green状态。

### 快速查看集群中有哪些索引

GET /\_cat/indices?v

health status index uuid pri rep docs.count docs.deleted store.size pri.store.size

yellow open .kibana rUm9n9wMRQCCrRDEhqneBg 1 1 1 0 3.1kb 3.1kb

### 简单的索引操作

创建索引：PUT /test\_index?pretty

health status index uuid pri rep docs.count docs.deleted store.size pri.store.size

yellow open test\_index XmS9DTAtSkSZSwWhhGEKkQ 5 1 0 0 650b 650b

yellow open .kibana rUm9n9wMRQCCrRDEhqneBg 1 1 1 0 3.1kb 3.1kb

删除索引：DELETE /test\_index?pretty

health status index uuid pri rep docs.count docs.deleted store.size pri.store.size

yellow open .kibana rUm9n9wMRQCCrRDEhqneBg 1 1 1 0 3.1kb 3.1kb

### 商品的CRUD操作

#### 新增商品：新增文档，建立索引

PUT /index/type/id

{

"json数据"

}

PUT /ecommerce/product/1

{

"name" : "gaolujie yagao",

"desc" : "gaoxiao meibai",

"price" : 30,

"producer" : "gaolujie producer",

"tags": [ "meibai", "fangzhu" ]

}

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

PUT /ecommerce/product/2

{

"name" : "jiajieshi yagao",

"desc" : "youxiao fangzhu",

"price" : 25,

"producer" : "jiajieshi producer",

"tags": [ "fangzhu" ]

}

PUT /ecommerce/product/3

{

"name" : "zhonghua yagao",

"desc" : "caoben zhiwu",

"price" : 40,

"producer" : "zhonghua producer",

"tags": [ "qingxin" ]

}

es会自动建立index和type，不需要提前创建，而且es默认会对document每个field都建立倒排索引，让其可以被搜索

#### 查询商品：检索文档

GET /index/type/id

GET /ecommerce/product/1

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 1,

"found": true,

"\_source": {

"name": "gaolujie yagao",

"desc": "gaoxiao meibai",

"price": 30,

"producer": "gaolujie producer",

"tags": [

"meibai",

"fangzhu"

]

}

}

#### 修改商品：替换文档

PUT /ecommerce/product/1

{

"name" : "jiaqiangban gaolujie yagao",

"desc" : "gaoxiao meibai",

"price" : 30,

"producer" : "gaolujie producer",

"tags": [ "meibai", "fangzhu" ]

}

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 2,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

PUT /ecommerce/product/1

{

"name" : "jiaqiangban gaolujie yagao"

}

替换方式有一个不好，即使必须带上所有的field，才能去进行信息的修改

#### 修改商品：更新文档

POST /ecommerce/product/1/\_update

{

"doc": {

"name": "jiaqiangban gaolujie yagao"

}

}

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 8,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

#### 删除商品：删除文档

DELETE /ecommerce/product/1

{

"found": true,

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 9,

"result": "deleted",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"found": false

}

## 多种查询方式

### 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

{

"took": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 1,

"hits": [

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "2",

"\_score": 1,

"\_source": {

"name": "jiajieshi yagao",

"desc": "youxiao fangzhu",

"price": 25,

"producer": "jiajieshi producer",

"tags": [

"fangzhu"

]

}

},

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_score": 1,

"\_source": {

"name": "gaolujie yagao",

"desc": "gaoxiao meibai",

"price": 30,

"producer": "gaolujie producer",

"tags": [

"meibai",

"fangzhu"

]

}

},

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "3",

"\_score": 1,

"\_source": {

"name": "zhonghua yagao",

"desc": "caoben zhiwu",

"price": 40,

"producer": "zhonghua producer",

"tags": [

"qingxin"

]

}

}

]

}

}

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

### 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

查询所有的商品

GET /ecommerce/product/\_search

{

"query": { "match\_all": {} }

}

查询名称包含yagao的商品，同时按照价格降序排序

GET /ecommerce/product/\_search

{

"query" : {

"match" : {

"name" : "yagao"

}

},

"sort": [

{ "price": "desc" }

]

}

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

GET /ecommerce/product/\_search

{

"query": { "match\_all": {} },

"from": 1,

"size": 1

}

指定要查询出来商品的名称和价格就可以

GET /ecommerce/product/\_search

{

"query": { "match\_all": {} },

"\_source": ["name", "price"]

}

更加适合生产环境的使用，可以构建复杂的查询

### query filter

搜索商品名称包含yagao，而且售价大于25元的商品

GET /ecommerce/product/\_search

{

"query" : {

"bool" : {

"must" : {

"match" : {

"name" : "yagao"

}

},

"filter" : {

"range" : {

"price" : { "gt" : 25 }

}

}

}

}

}

### full-text search（全文检索）

GET /ecommerce/product/\_search

{

"query" : {

"match" : {

"producer" : "yagao producer"

}

}

}

尽量，无论是学什么技术，比如说你当初学java，学linux，学shell，学javascript，学hadoop。。。。一定自己动手，特别是手工敲各种命令和代码，切记切记，减少复制粘贴的操作。只有自己动手手工敲，学习效果才最好。

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

{

"took": 4,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 4,

"max\_score": 0.70293105,

"hits": [

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "4",

"\_score": 0.70293105,

"\_source": {

"name": "special yagao",

"desc": "special meibai",

"price": 50,

"producer": "special yagao producer",

"tags": [

"meibai"

]

}

},

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_score": 0.25811607,

"\_source": {

"name": "gaolujie yagao",

"desc": "gaoxiao meibai",

"price": 30,

"producer": "gaolujie producer",

"tags": [

"meibai",

"fangzhu"

]

}

},

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "3",

"\_score": 0.25811607,

"\_source": {

"name": "zhonghua yagao",

"desc": "caoben zhiwu",

"price": 40,

"producer": "zhonghua producer",

"tags": [

"qingxin"

]

}

},

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "2",

"\_score": 0.1805489,

"\_source": {

"name": "jiajieshi yagao",

"desc": "youxiao fangzhu",

"price": 25,

"producer": "jiajieshi producer",

"tags": [

"fangzhu"

]

}

}

]

}

}

### 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

GET /ecommerce/product/\_search

{

"query" : {

"match\_phrase" : {

"producer" : "yagao producer"

}

}

}

{

"took": 11,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 0.70293105,

"hits": [

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "4",

"\_score": 0.70293105,

"\_source": {

"name": "special yagao",

"desc": "special meibai",

"price": 50,

"producer": "special yagao producer",

"tags": [

"meibai"

]

}

}

]

}

}

### highlight search（高亮搜索结果）

GET /ecommerce/product/\_search

{

"query" : {

"match" : {

"producer" : "producer"

}

},

"highlight": {

"fields" : {

"producer" : {}

}

}

}

## 嵌套聚合，下钻分析，聚合分析

第一个分析需求：计算每个tag下的商品数量

GET /ecommerce/product/\_search

{

"aggs": {

"group\_by\_tags": {

"terms": { "field": "tags" }

}

}

}

将文本field的fielddata属性设置为true

PUT /ecommerce/\_mapping/product

{

"properties": {

"tags": {

"type": "text",

"fielddata": true

}

}

}

GET /ecommerce/product/\_search

{

"size": 0,

"aggs": {

"all\_tags": {

"terms": { "field": "tags" }

}

}

}

{

"took": 20,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 4,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_tags": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "fangzhu",

"doc\_count": 2

},

{

"key": "meibai",

"doc\_count": 2

},

{

"key": "qingxin",

"doc\_count": 1

}

]

}

}

}

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

{

"size": 0,

"query": {

"match": {

"name": "yagao"

}

},

"aggs": {

"all\_tags": {

"terms": {

"field": "tags"

}

}

}

}

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

{

"size": 0,

"aggs" : {

"group\_by\_tags" : {

"terms" : { "field" : "tags" },

"aggs" : {

"avg\_price" : {

"avg" : { "field" : "price" }

}

}

}

}

}

{

"took": 8,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 4,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_tags": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "fangzhu",

"doc\_count": 2,

"avg\_price": {

"value": 27.5

}

},

{

"key": "meibai",

"doc\_count": 2,

"avg\_price": {

"value": 40

}

},

{

"key": "qingxin",

"doc\_count": 1,

"avg\_price": {

"value": 40

}

}

]

}

}

}

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

{

"size": 0,

"aggs" : {

"all\_tags" : {

"terms" : { "field" : "tags", "order": { "avg\_price": "desc" } },

"aggs" : {

"avg\_price" : {

"avg" : { "field" : "price" }

}

}

}

}

}

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

{

"size": 0,

"aggs": {

"group\_by\_price": {

"range": {

"field": "price",

"ranges": [

{

"from": 0,

"to": 20

},

{

"from": 20,

"to": 40

},

{

"from": 40,

"to": 50

}

]

},

"aggs": {

"group\_by\_tags": {

"terms": {

"field": "tags"

},

"aggs": {

"average\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

}

}

## Elasticsearch的基础分布式架构

1. Elasticsearch对复杂分布式机制的透明隐藏特性

Elasticsearch是一套分布式的系统，分布式是为了应对大数据量

隐藏了复杂的分布式机制

分片机制（我们之前随随便便就将一些document插入到es集群中去了，我们有没有关注过数据怎么进行分片的，数据到哪个shard中去）

cluster discovery（集群发现机制，我们之前在做那个集群status从yellow转green的实验里，直接启动了第二个es进程，那个进程作为一个node自动就发现了集群，并且加入了进去，还接受了部分数据，replica shard）

shard负载均衡（举例，假设现在有3个节点，总共有25个shard要分配到3个节点上去，es会自动进行均匀分配，以保持每个节点的均衡的读写负载请求）

shard副本，请求路由，集群扩容，shard重分配

1. Elasticsearch的垂直扩容与水平扩容

垂直扩容：采购更强大的服务器，成本非常高昂，而且会有瓶颈，假设世界上最强大的服务器容量就是10T，但是当你的总数据量达到5000T的时候，你要采购多少台最强大的服务器啊

水平扩容：业界经常采用的方案，采购越来越多的普通服务器，性能比较一般，但是很多普通服务器组织在一起，就能构成强大的计算和存储能力

普通服务器：1T，1万，100万

强大服务器：10T，50万，500万

扩容对应用程序的透明性

1. 增减或减少节点时的数据rebalance

保持负载均衡

1. master节点

（1）创建或删除索引

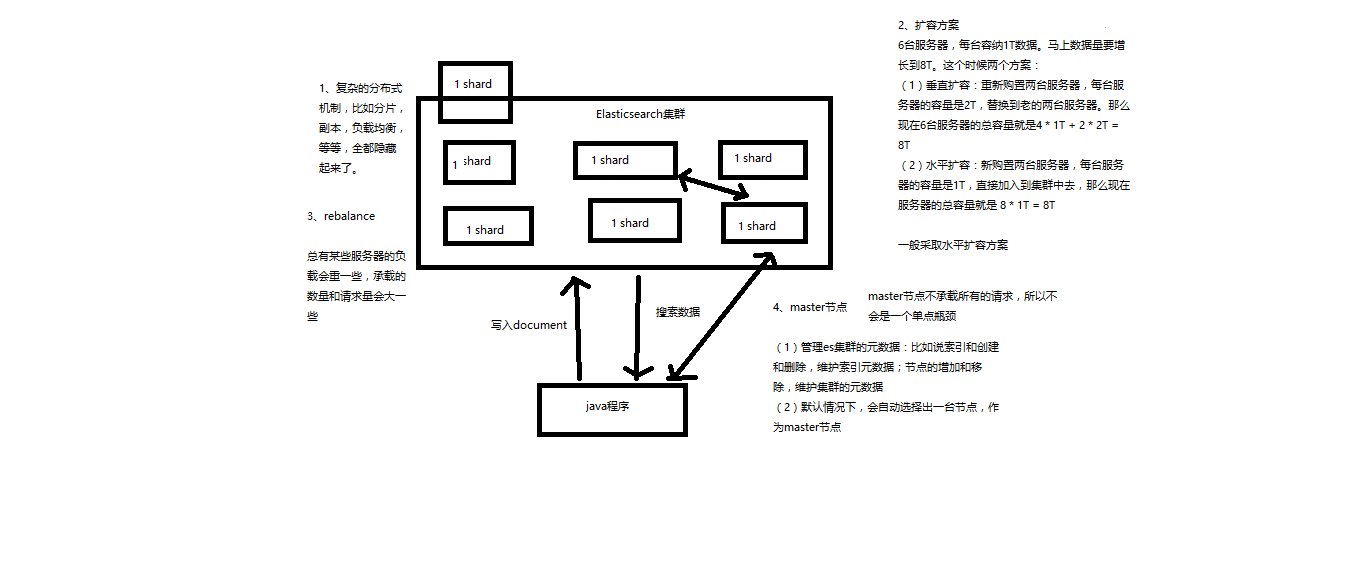
（2）增加或删除节点

1. 节点平等的分布式架构

（1）节点对等，每个节点都能接收所有的请求

（2）自动请求路由

（3）响应收集



**ES的基础分布式架构**

## shard&replica机制再次梳理以及单node环境中创建index图解

### shard&replica机制梳理

* 1. index包含多个shard
  2. 每个shard都是一个最小工作单元，承载部分数据，lucene实例，完整的建立索引和处理请求的能力
  3. 增减节点时，shard会自动在nodes中负载均衡
  4. primary shard和replica shard，每个document肯定只存在于某一个primary shard以及其对应的replica shard中，不可能存在于多个primary shard
  5. replica shard是primary shard的副本，负责容错，以及承担读请求负载
  6. primary shard的数量在创建索引的时候就固定了，replica shard的数量可以随时修改
  7. primary shard的默认数量是5，replica默认是1，默认有10个shard，5个primary shard，5个replica shard
  8. primary shard不能和自己的replica shard放在同一个节点上（否则节点宕机，primary shard和副本都丢失，起不到容错的作用），但是可以和其他primary shard的replica shard放在同一个节点上

### 单node环境下创建index是什么样子的

1. 单node环境下，创建一个index，有3个primary shard，3个replica shard
2. 集群status是yellow
3. 这个时候，只会将3个primary shard分配到仅有的一个node上去，另外3个replica shard是无法分配的
4. 集群可以正常工作，但是一旦出现节点宕机，数据全部丢失，而且集群不可用，无法承接任何请求

PUT /test\_index

{

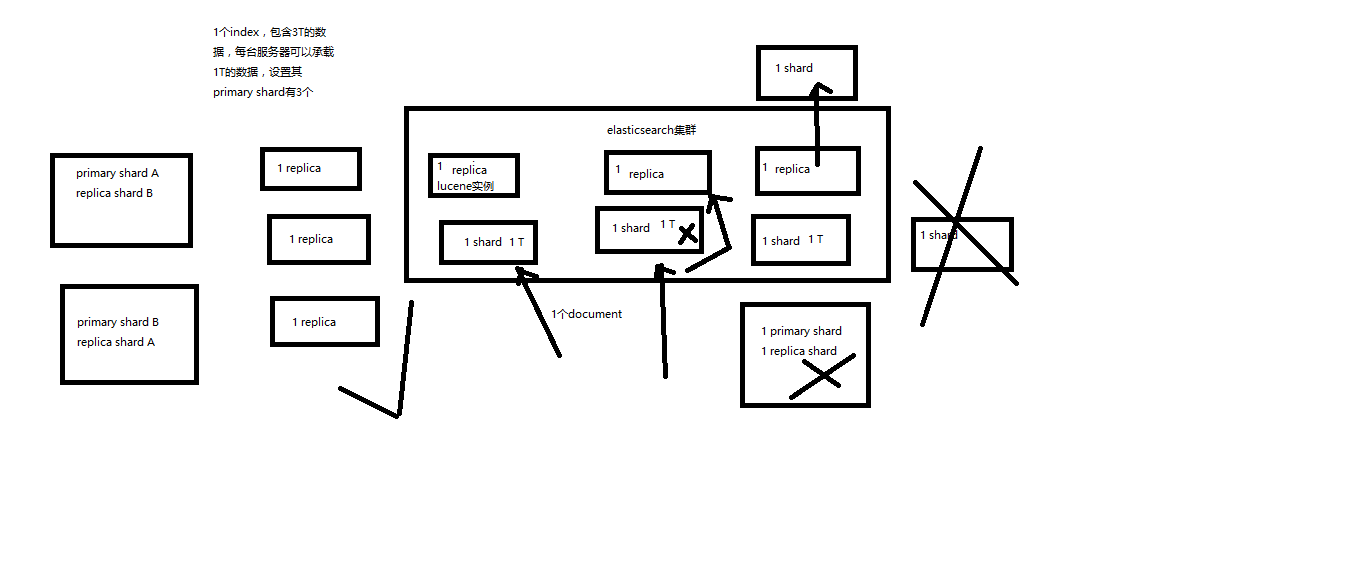
"settings" : {

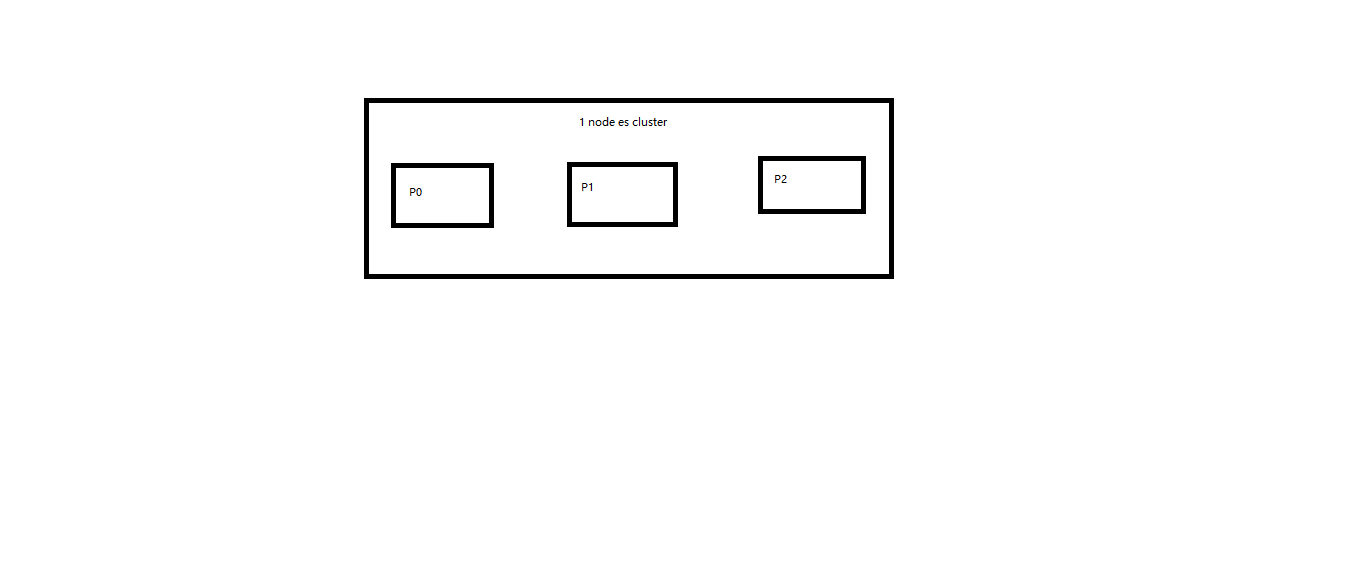
"number\_of\_shards" : 3,

"number\_of\_replicas" : 1

}

}

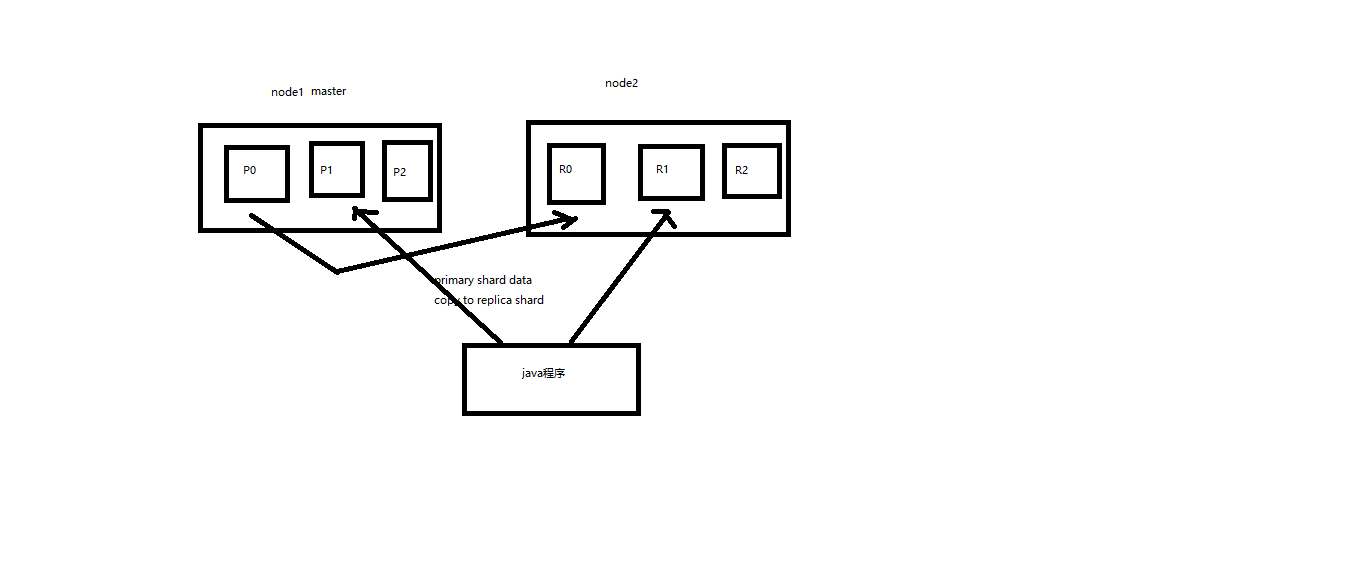
**shard&replica机制**



**单node环境下创建index**

## 2个node环境下replica shard是如何分配的

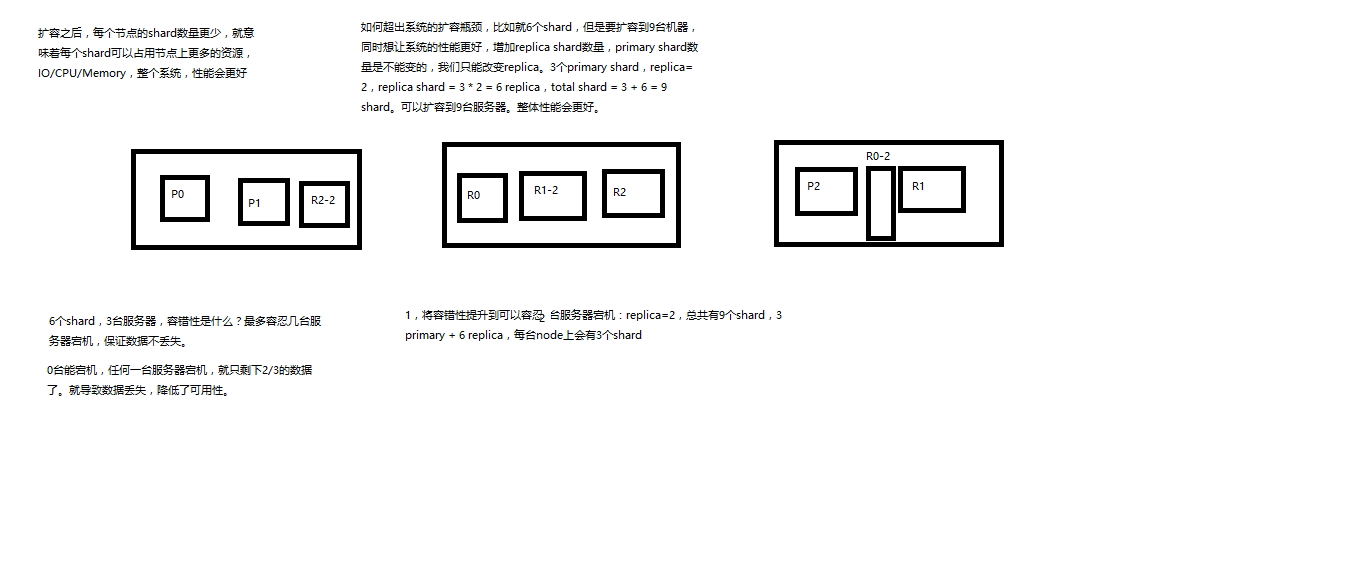
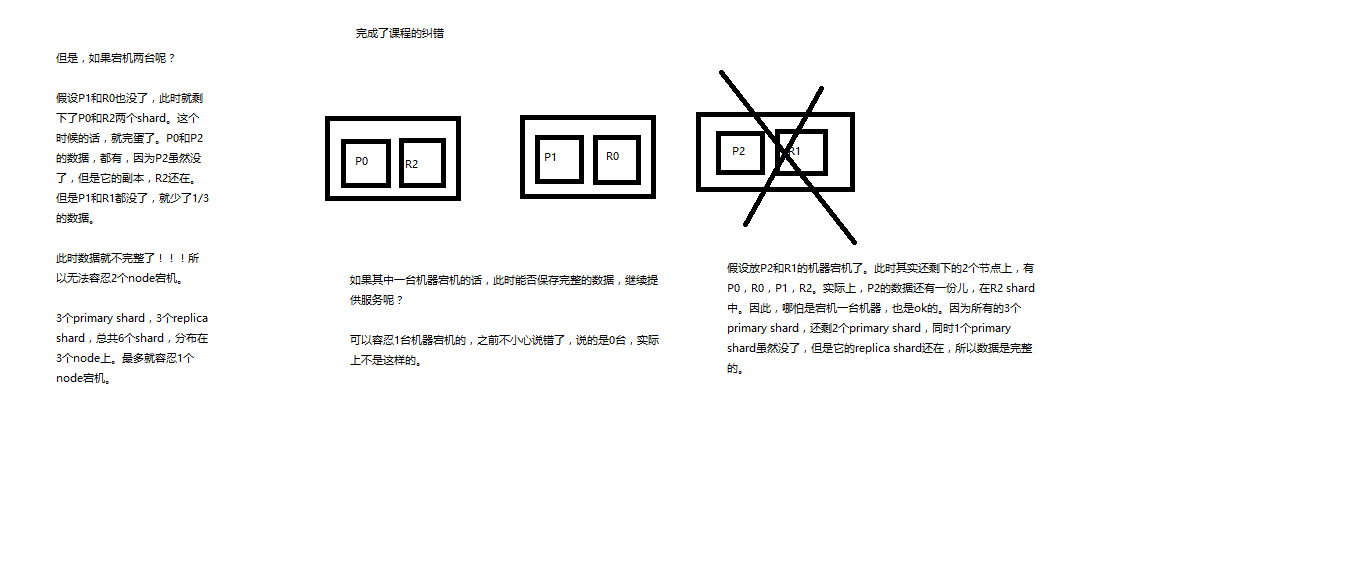
* 1. replica shard分配：3个primary shard，3个replica shard，1 node
  2. primary ---> replica同步
  3. 读请求：primary/replica



**2个node环境下replica shard是如何分配的**

## 横向扩容过程，如何超出扩容极限，以及如何提升容错性

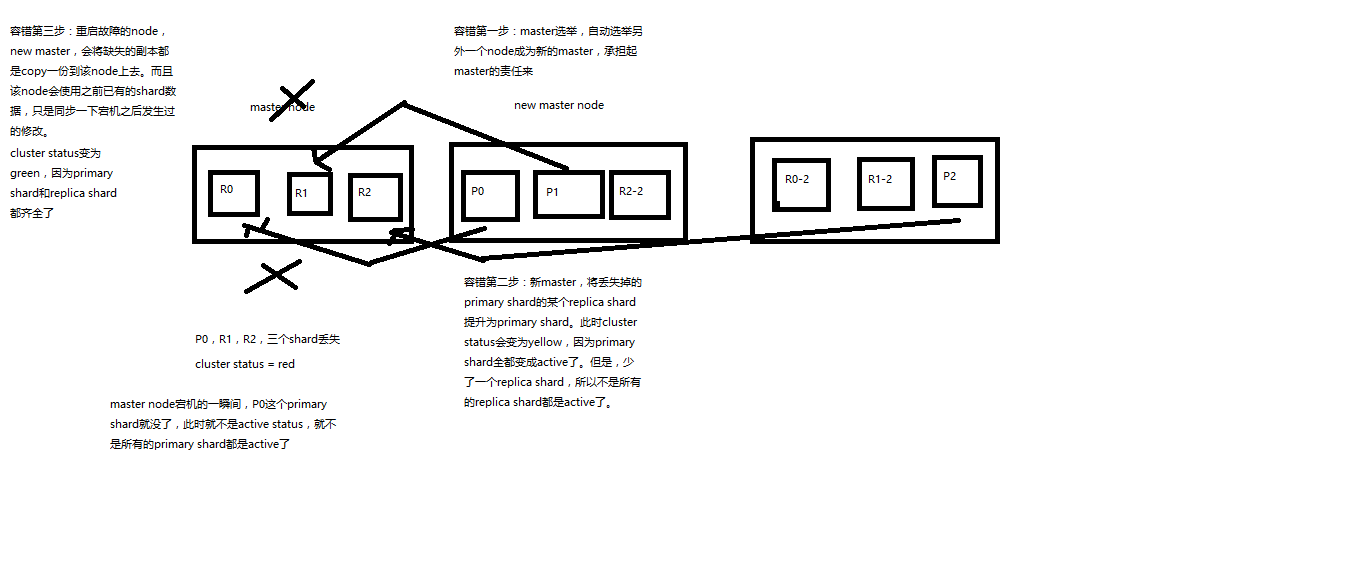
1. primary&replica自动负载均衡，6个shard，3 primary，3 replica
2. 每个node有更少的shard，IO/CPU/Memory资源给每个shard分配更多，每个shard性能更好
3. 扩容的极限，6个shard（3 primary，3 replica），最多扩容到6台机器，每个shard可以占用单台服务器的所有资源，性能最好
4. 超出扩容极限，动态修改replica数量，9个shard（3primary，6 replica），扩容到9台机器，比3台机器时，拥有3倍的读吞吐量
5. 3台机器下，9个shard（3 primary，6 replica），资源更少，但是容错性更好，最多容纳2台机器宕机，6个shard只能容纳0台机器宕机
6. 综合起来看，就是说，一方面讲扩容的原理，怎么扩容，怎么提升系统整体吞吐量；另一方面要考虑到系统的容错性，怎么保证提高容错性，让尽可能多的服务器宕机，保证数据不丢失

**扩容过程分析**

**容错纠正**

## Elasticsearch容错机制：master选举，replica容错，数据恢复

1. 9 shard，3 node
2. master node宕机，自动master选举，red
3. replica容错：新master将replica提升为primary shard，yellow
4. 重启宕机node，master copy replica到该node，使用原有的shard并同步宕机后的修改，green



**es容错过程分析**

## document的核心元数据以及index创建反例

### \_index元数据

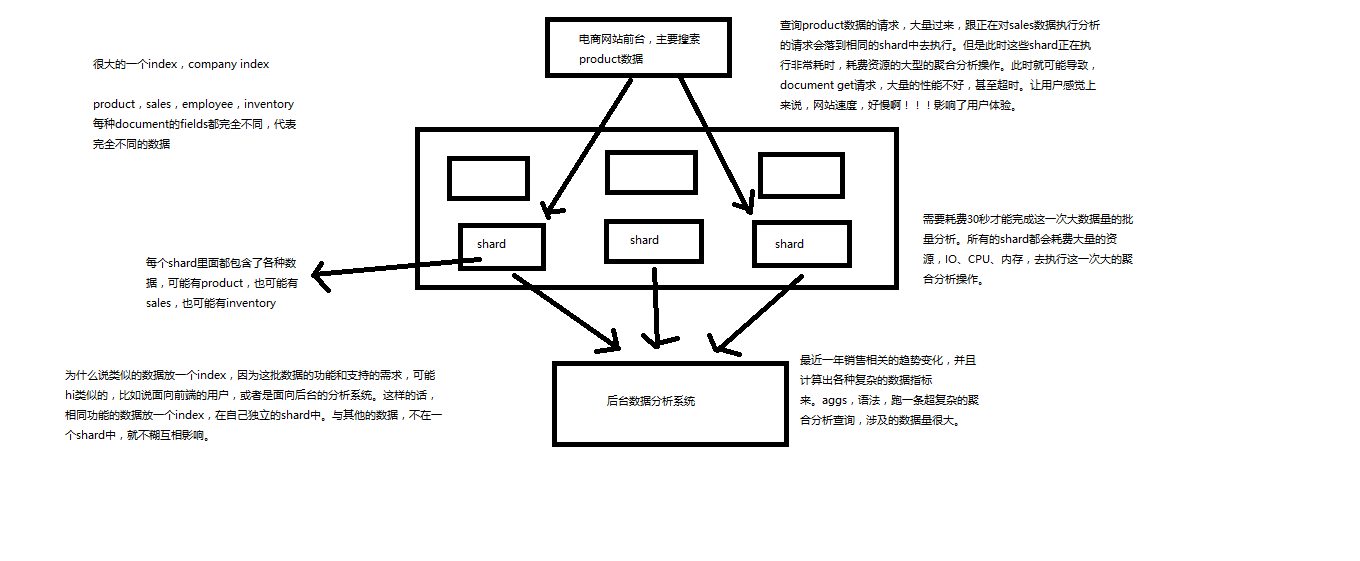
* 1. 代表一个document存放在哪个index中
  2. 类似的数据放在一个索引，非类似的数据放不同索引：product index（包含了所有的商品），sales index（包含了所有的商品销售数据），inventory index（包含了所有库存相关的数据）。如果你把比如product，sales，human resource（employee），全都放在一个大的index里面，比如说company index，不合适的。
  3. index中包含了很多类似的document：类似是什么意思，其实指的就是说，这些document的fields很大一部分是相同的，你说你放了3个document，每个document的fields都完全不一样，这就不是类似了，就不太适合放到一个index里面去了。
  4. 索引名称必须是小写的，不能用下划线开头，不能包含逗号：product，website，blog

### type元数据

1. 代表document属于index中的哪个类别（type）
2. 一个索引通常会划分为多个type，逻辑上对index中有些许不同的几类数据进行分类：因为一批相同的数据，可能有很多相同的fields，但是还是可能会有一些轻微的不同，可能会有少数fields是不一样的，举个例子，就比如说，商品，可能划分为电子商品，生鲜商品，日化商品，等等。
3. type名称可以是大写或者小写，但是同时不能用下划线开头，不能包含逗号

### \_id元数据

1. 代表document的唯一标识，与index和type一起，可以唯一标识和定位一个document
2. （2）我们可以手动指定document的id（put /index/type/id），也可以不指定，由es自动为我们创建一个id



**index如何创建的反例分析**

## document id的手动指定与自动生成两种方式解析

### 手动指定document id

#### 根据应用情况来说，是否满足手动指定document id的前提

一般来说，是从某些其他的系统中，导入一些数据到es时，会采取这种方式，就是使用系统中已有数据的唯一标识，作为es中document的id。举个例子，比如说，我们现在在开发一个电商网站，做搜索功能，或者是OA系统，做员工检索功能。这个时候，数据首先会在网站系统或者IT系统内部的数据库中，会先有一份，此时就肯定会有一个数据库的primary key（自增长，UUID，或者是业务编号）。如果将数据导入到es中，此时就比较适合采用数据在数据库中已有的primary key。

如果说，我们是在做一个系统，这个系统主要的数据存储就是es一种，也就是说，数据产生出来以后，可能就没有id，直接就放es一个存储，那么这个时候，可能就不太适合说手动指定document id的形式了，因为你也不知道id应该是什么，此时可以采取让es自动生成id的方式。

#### put /index/type/id

PUT /test\_index/test\_type/2

{

"test\_content": "my test"

}

### 自动生成document id

post /index/type

POST /test\_index/test\_type

{

"test\_content": "my test"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "AVp4RN0bhjxldOOnBxaE",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

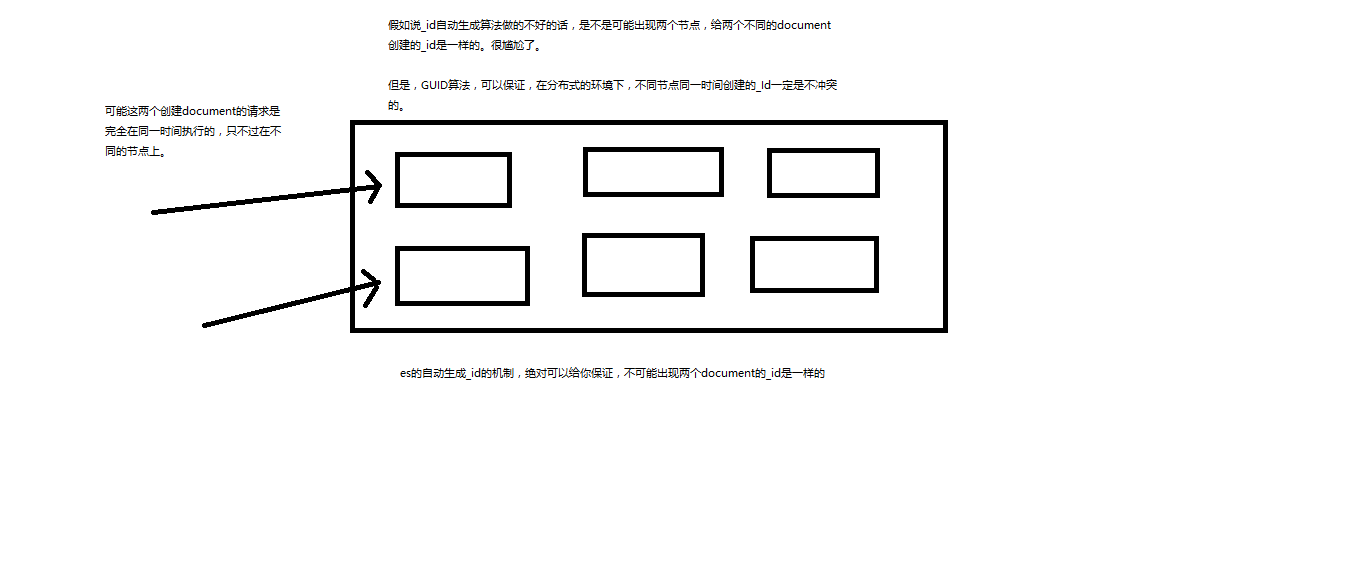
"failed": 0

},

"created": true

}

自动生成的id，长度为20个字符，URL安全，base64编码，GUID，分布式系统并行生成时不可能会发生冲突

**GUID不冲突解释**

## document的\_source元数据以及定制返回结果

### \_source元数据

put /test\_index/test\_type/1

{

"test\_field1": "test field1",

"test\_field2": "test field2"

}

get /test\_index/test\_type/1

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "1",

"\_version": 2,

"found": true,

"\_source": {

"test\_field1": "test field1",

"test\_field2": "test field2"

}

}

\_source元数据：就是说，我们在创建一个document的时候，使用的那个放在request body中的json串，默认情况下，在get的时候，会原封不动的给我们返回回来。

### 定制返回结果

定制返回的结果，指定\_source中，返回哪些field

GET /test\_index/test\_type/1?\_source=test\_field1,test\_field2

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "1",

"\_version": 2,

"found": true,

"\_source": {

"test\_field2": "test field2"

}

}

## document的全量替换、强制创建以及图解lazy delete机制

### document的全量替换

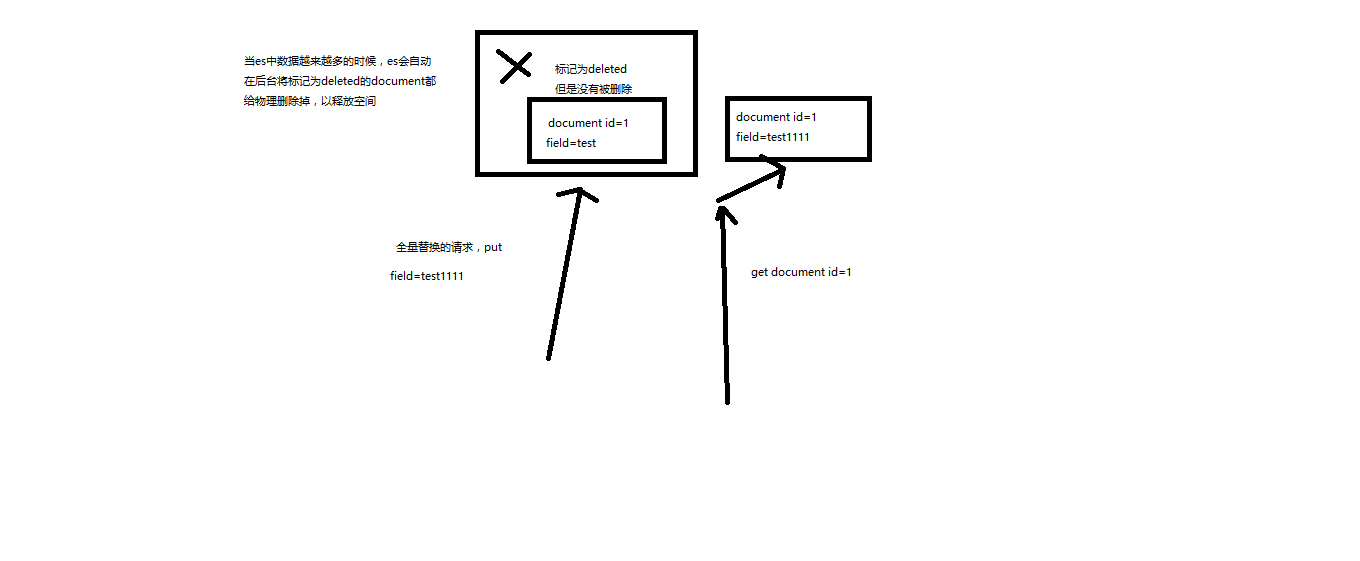
* 1. 语法与创建文档是一样的，如果document id不存在，那么就是创建；如果document id已经存在，那么就是全量替换操作，替换document的json串内容
  2. document是不可变的，如果要修改document的内容，第一种方式就是全量替换，直接对document重新建立索引，替换里面所有的内容
  3. es会将老的document标记为deleted，然后新增我们给定的一个document，当我们创建越来越多的document的时候，es会在适当的时机在后台自动删除标记为deleted的document

### document的强制创建

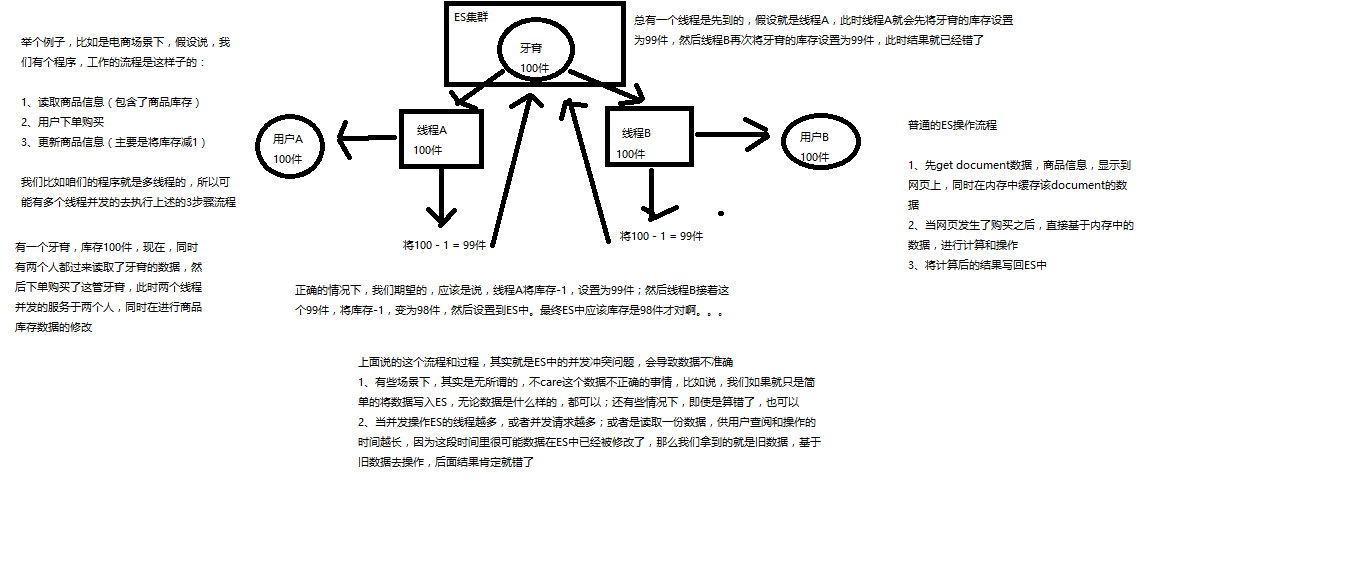
1. 创建文档与全量替换的语法是一样的，有时我们只是想新建文档，不想替换文档，如果强制进行创建呢？
2. PUT /index/type/id?op\_type=create，PUT /index/type/id/\_create

### 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

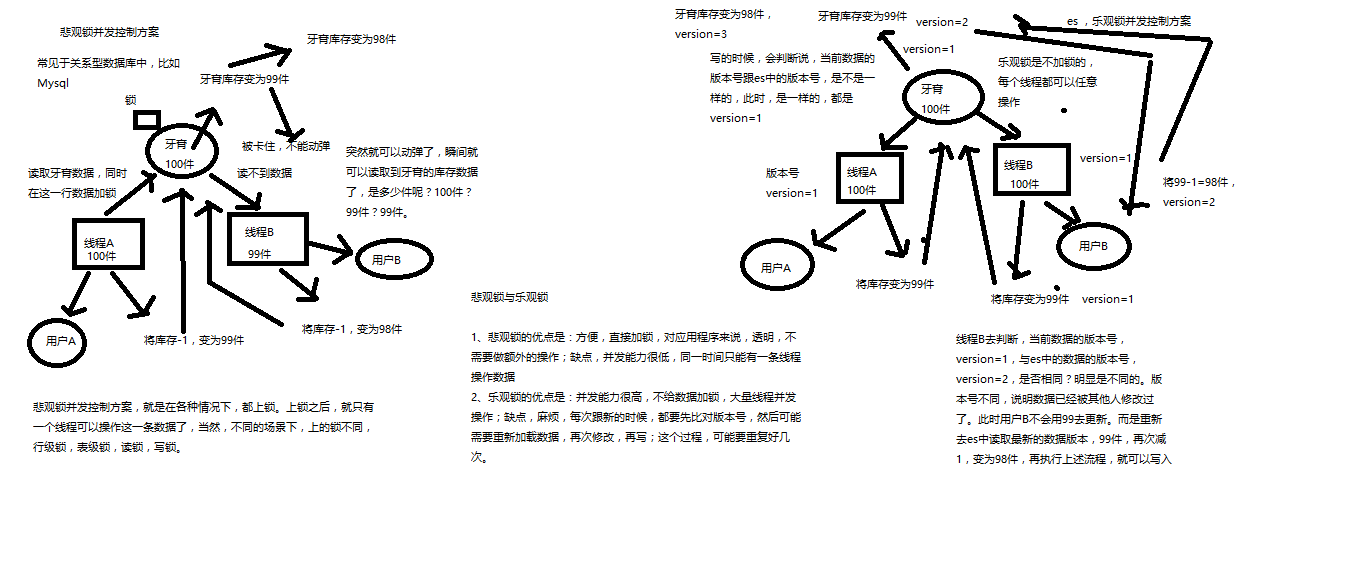
**es document delete的原理**

## Elasticsearch并发冲突问题



**剖析Elasticsearch并发冲突问题**

## 悲观锁与乐观锁两种并发控制方案



**悲观锁与乐观锁两种并发控制方案**

## 如何基于\_version进行乐观锁并发控制

### \_version元数据

PUT /test\_index/test\_type/6

{

"test\_field": "test test"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "6",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

第一次创建一个document的时候，它的\_version内部版本号就是1；以后，每次对这个document执行修改或者删除操作，都会对这个\_version版本号自动加1；哪怕是删除，也会对这条数据的版本号加1

{

"found": true,

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "6",

"\_version": 4,

"result": "deleted",

"\_shards": {

"total": 2,

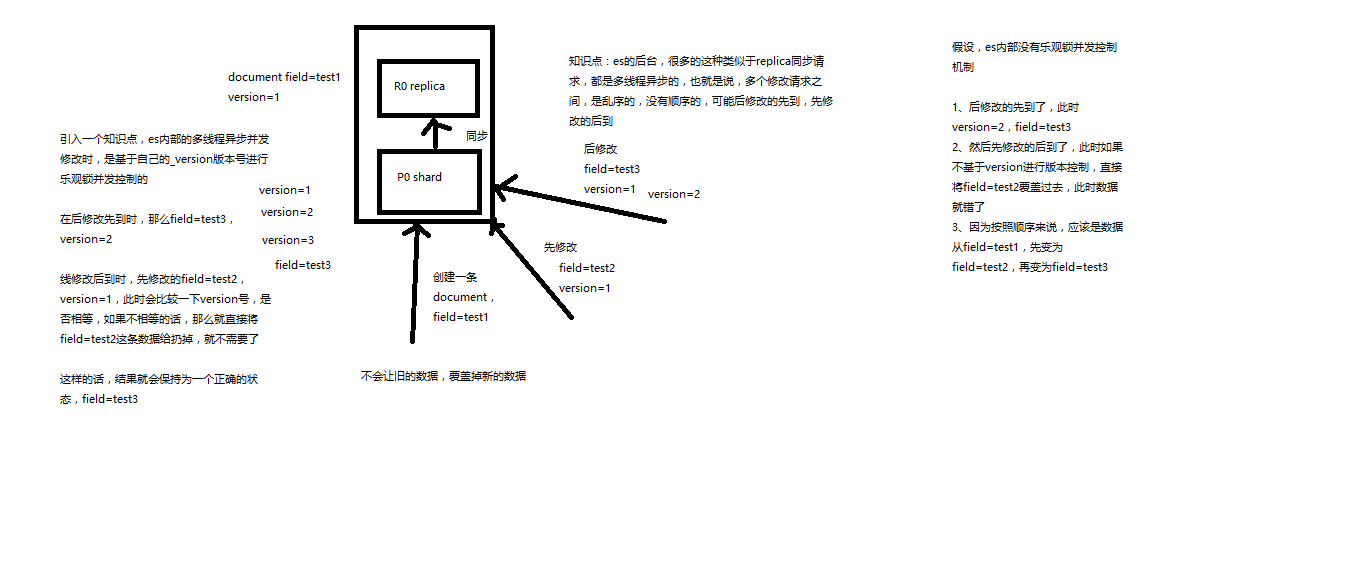
"successful": 1,

"failed": 0

}

}

可以发现，在删除一个document之后，可以从一个侧面证明，它不是立即物理删除掉的，因为它的一些版本号等信息还是保留着的。先删除一条document，再重新创建这条document，其实会在delete version基础之上，再把version号加1

**Elasticsearch内部如何基于\_version进行乐观锁并发控制**

## \_version进行乐观锁并发控制

1. 先构造一条数据出来

PUT /test\_index/test\_type/7

{

"test\_field": "test test"

}

1. 模拟两个客户端，都获取到了同一条数据

GET test\_index/test\_type/7

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "7",

"\_version": 1,

"found": true,

"\_source": {

"test\_field": "test test"

}

}

1. 其中一个客户端，先更新了一下这个数据

同时带上数据的版本号，确保说，es中的数据的版本号，跟客户端中的数据的版本号是相同的，才能修改

PUT /test\_index/test\_type/7?version=1

{

"test\_field": "test client 1"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "7",

"\_version": 2,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

1. 另外一个客户端，尝试基于version=1的数据去进行修改，同样带上version版本号，进行乐观锁的并发控制

PUT /test\_index/test\_type/7?version=1

{

"test\_field": "test client 2"

}

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][7]: version conflict, current version [2] is different than the one provided [1]",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "3",

"index": "test\_index"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][7]: version conflict, current version [2] is different than the one provided [1]",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "3",

"index": "test\_index"

},

"status": 409

}

1. 在乐观锁成功阻止并发问题之后，尝试正确的完成更新

GET /test\_index/test\_type/7

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "7",

"\_version": 2,

"found": true,

"\_source": {

"test\_field": "test client 1"

}

}

基于最新的数据和版本号，去进行修改，修改后，带上最新的版本号，可能这个步骤会需要反复执行好几次，才能成功，特别是在多线程并发更新同一条数据很频繁的情况下

PUT /test\_index/test\_type/7?version=2

{

"test\_field": "test client 2"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "7",

"\_version": 3,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

## external version进行乐观锁并发控制

external version

es提供了一个feature，就是说，你可以不用它提供的内部\_version版本号来进行并发控制，可以基于你自己维护的一个版本号来进行并发控制。举个列子，加入你的数据在mysql里也有一份，然后你的应用系统本身就维护了一个版本号，无论是什么自己生成的，程序控制的。这个时候，你进行乐观锁并发控制的时候，可能并不是想要用es内部的\_version来进行控制，而是用你自己维护的那个version来进行控制。

?version=1

?version=1&version\_type=external

version\_type=external，唯一的区别在于，\_version，只有当你提供的version与es中的\_version一模一样的时候，才可以进行修改，只要不一样，就报错；当version\_type=external的时候，只有当你提供的version比es中的\_version大的时候，才能完成修改

es，\_version=1，?version=1，才能更新成功

es，\_version=1，?version>1&version\_type=external，才能成功，比如说?version=2&version\_type=external

* + 1. 先构造一条数据

PUT /test\_index/test\_type/8

{

"test\_field": "test"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

* + 1. 模拟两个客户端同时查询到这条数据

GET /test\_index/test\_type/8

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 1,

"found": true,

"\_source": {

"test\_field": "test"

}

}

* + 1. 第一个客户端先进行修改，此时客户端程序是在自己的数据库中获取到了这条数据的最新版本号，比如说是2

PUT /test\_index/test\_type/8?version=2&version\_type=external

{

"test\_field": "test client 1"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 2,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

* + 1. 模拟第二个客户端，同时拿到了自己数据库中维护的那个版本号，也是2，同时基于version=2发起了修改

PUT /test\_index/test\_type/8?version=2&version\_type=external

{

"test\_field": "test client 2"

}

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][8]: version conflict, current version [2] is higher or equal to the one provided [2]",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "1",

"index": "test\_index"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][8]: version conflict, current version [2] is higher or equal to the one provided [2]",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "1",

"index": "test\_index"

},

"status": 409

}

* + 1. 在并发控制成功后，重新基于最新的版本号发起更新

GET /test\_index/test\_type/8

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 2,

"found": true,

"\_source": {

"test\_field": "test client 1"

}

}

PUT /test\_index/test\_type/8?version=3&version\_type=external

{

"test\_field": "test client 2"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 3,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

## partial update实现原理

### 什么是partial update？

PUT /index/type/id，创建文档&替换文档，就是一样的语法

一般对应到应用程序中，每次的执行流程基本是这样的：

* + 1. 应用程序先发起一个get请求，获取到document，展示到前台界面，供用户查看和修改
    2. 用户在前台界面修改数据，发送到后台
    3. 后台代码，会将用户修改的数据在内存中进行执行，然后封装好修改后的全量数据
    4. 然后发送PUT请求，到es中，进行全量替换
    5. es将老的document标记为deleted，然后重新创建一个新的document

partial update

post /index/type/id/\_update

{

"doc": {

"要修改的少数几个field即可，不需要全量的数据"

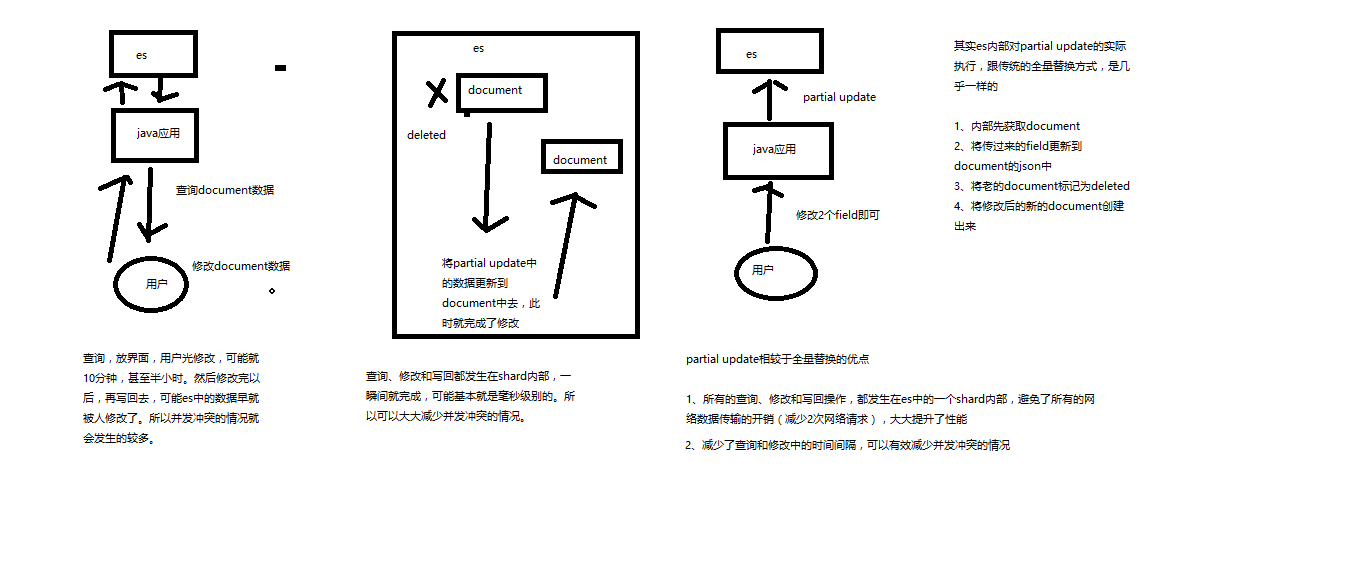
}

}

看起来，好像就比较方便了，每次就传递少数几个发生修改的field即可，不需要将全量的document数据发送过去

### partial update实现原理以及其优点

partial update，看起来很方便的操作，实际内部的原理是什么样子的，然后它的优点是什么？



**partial update实现原理以及其优点**

如：

PUT /test\_index/test\_type/10

{

"test\_field1": "test1",

"test\_field2": "test2"

}

POST /test\_index/test\_type/10/\_update

{

"doc": {

"test\_field2": "updated test2"

}

}

## 基于groovy脚本进行partial update

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，如何执行partial update

es scripting module，我们会在高手进阶篇去讲解，这里就只是初步讲解一下

PUT /test\_index/test\_type/11

{

"num": 0,

"tags": []

}

* + 1. 内置脚本

POST /test\_index/test\_type/11/\_update

{

"script" : "ctx.\_source.num+=1"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "11",

"\_version": 2,

"found": true,

"\_source": {

"num": 1,

"tags": []

}

}

* + 1. 外部脚本

ctx.\_source.tags+=new\_tag

POST /test\_index/test\_type/11/\_update

{

"script": {

"lang": "groovy",

"file": "test-add-tags",

"params": {

"new\_tag": "tag1"

}

}

}

* + 1. 用脚本删除文档

ctx.op = ctx.\_source.num == count ? 'delete' : 'none'

POST /test\_index/test\_type/11/\_update

{

"script": {

"lang": "groovy",

"file": "test-delete-document",

"params": {

"count": 1

}

}

}

* + 1. 操作

POST /test\_index/test\_type/11/\_update

{

"doc": {

"num": 1

}

}

{

"error": {

"root\_cause": [

{

"type": "document\_missing\_exception",

"reason": "[test\_type][11]: document missing",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "4",

"index": "test\_index"

}

],

"type": "document\_missing\_exception",

"reason": "[test\_type][11]: document missing",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "4",

"index": "test\_index"

},

"status": 404

}

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": []

}

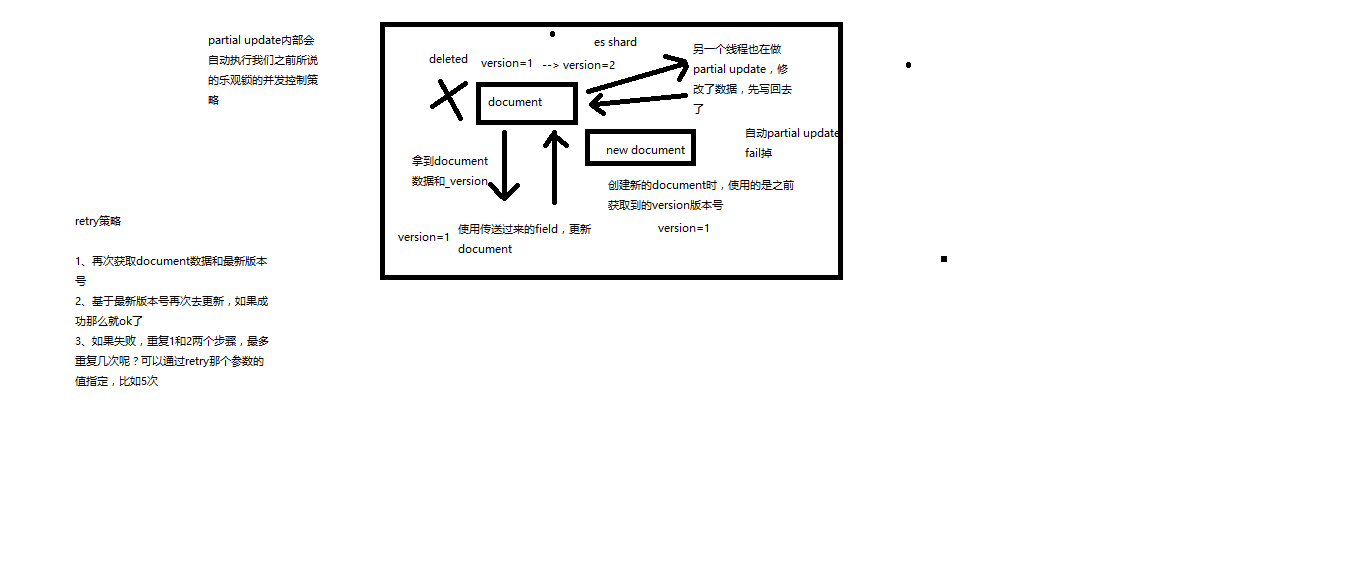
}

## partial update乐观锁并发控制原理以及相关操作

（1）partial update内置乐观锁并发控制

（2）retry\_on\_conflict重试机制

（3）\_version

post /index/type/id/\_update?retry\_on\_conflict=5&version=6

**partial update内置乐观锁并发控制**

## mget批量查询api

### 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

### mget的语法

#### 一条一条的查询

GET /test\_index/test\_type/1

GET /test\_index/test\_type/2

#### mget批量查询

GET /\_mget

{

"docs" : [

{

"\_index" : "test\_index",

"\_type" : "test\_type",

"\_id" : 1

},

{

"\_index" : "test\_index",

"\_type" : "test\_type",

"\_id" : 2

}

]

}

{

"docs": [

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "1",

"\_version": 2,

"found": true,

"\_source": {

"test\_field1": "test field1",

"test\_field2": "test field2"

}

},

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "2",

"\_version": 1,

"found": true,

"\_source": {

"test\_content": "my test"

}

}

]

}

#### 如果查询的document是一个index下的不同type种的话

GET /test\_index/\_mget

{

"docs" : [

{

"\_type" : "test\_type",

"\_id" : 1

},

{

"\_type" : "test\_type",

"\_id" : 2

}

]

}

#### 如果查询的数据都在同一个index下的同一个type下，最简单了

GET /test\_index/test\_type/\_mget

{

"ids": [1, 2]

}

#### 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

## bulk批量增删改

### bulk语法

POST /\_bulk

{ "delete": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "3" }}

{ "create": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "12" }}

{ "test\_field": "test12" }

{ "index": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "2" }}

{ "test\_field": "replaced test2" }

{ "update": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "1", "\_retry\_on\_conflict" : 3} }

{ "doc" : {"test\_field2" : "bulk test1"} }

每一个操作要两个json串，语法如下：

{"action": {"metadata"}}

{"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：

{"index": {"\_index": "test\_index", "\_type", "test\_type", "\_id": "1"}}

{"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

{

"error": {

"root\_cause": [

{

"type": "json\_e\_o\_f\_exception",

"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]"

}

],

"type": "json\_e\_o\_f\_exception",

"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]"

},

"status": 500

}

{

"took": 41,

"errors": true,

"items": [

{

"delete": {

"found": true,

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "10",

"\_version": 3,

"result": "deleted",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"status": 200

}

},

{

"create": {

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "3",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true,

"status": 201

}

},

{

"create": {

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "2",

"status": 409,

"error": {

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][2]: version conflict, document already exists (current version [1])",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "2",

"index": "test\_index"

}

}

},

{

"index": {

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "4",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true,

"status": 201

}

},

{

"index": {

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "2",

"\_version": 2,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false,

"status": 200

}

},

{

"update": {

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "1",

"\_version": 3,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"status": 200

}

}

]

}

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

POST /test\_index/\_bulk

{ "delete": { "\_type": "test\_type", "\_id": "3" }}

{ "create": { "\_type": "test\_type", "\_id": "12" }}

{ "test\_field": "test12" }

{ "index": { "\_type": "test\_type" }}

{ "test\_field": "auto-generate id test" }

{ "index": { "\_type": "test\_type", "\_id": "2" }}

{ "test\_field": "replaced test2" }

{ "update": { "\_type": "test\_type", "\_id": "1", "\_retry\_on\_conflict" : 3} }

{ "doc" : {"test\_field2" : "bulk test1"} }

POST /test\_index/test\_type/\_bulk

{ "delete": { "\_id": "3" }}

{ "create": { "\_id": "12" }}

{ "test\_field": "test12" }

{ "index": { }}

{ "test\_field": "auto-generate id test" }

{ "index": { "\_id": "2" }}

{ "test\_field": "replaced test2" }

{ "update": { "\_id": "1", "\_retry\_on\_conflict" : 3} }

{ "doc" : {"test\_field2" : "bulk test1"} }

### 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。

## 什么是distributed document store

Elasticsearch在跑起来以后，其实起到的第一个最核心的功能，就是一个分布式的文档数据存储系统。ES是分布式的。文档数据存储系统。文档数据，存储系统。

文档数据：es可以存储和操作json文档类型的数据，而且这也是es的核心数据结构。

存储系统：es可以对json文档类型的数据进行存储，查询，创建，更新，删除，等等操作。其实已经起到了一个什么样的效果呢？其实ES满足了这些功能，就可以说已经是一个NoSQL的存储系统了。

围绕着document在操作，其实就是把es当成了一个NoSQL存储引擎，一个可以存储文档类型数据的存储系统，在操作里面的document。

es可以作为一个分布式的文档存储系统，所以说，我们的应用系统，是不是就可以基于这个概念，去进行相关的应用程序的开发了。

什么类型的应用程序呢？

1. 数据量较大，es的分布式本质，可以帮助你快速进行扩容，承载大量数据
2. 数据结构灵活多变，随时可能会变化，而且数据结构之间的关系，非常复杂，如果我们用传统数据库，那是不是很坑，因为要面临大量的表
3. 对数据的相关操作，较为简单，比如就是一些简单的增删改查，用我们之前讲解的那些document操作就可以搞定
4. NoSQL数据库，适用的也是类似于上面的这种场景

举个例子，比如说像一些网站系统，或者是普通的电商系统，博客系统，面向对象概念比较复杂，但是作为终端网站来说，没什么太复杂的功能，就是一些简单的CRUD操作，而且数据量可能还比较大。这个时候选用ES这种NoSQL型的数据存储，比传统的复杂的功能务必强大的支持SQL的关系型数据库，更加合适一些。无论是性能，还是吞吐量，可能都会更好。

## document数据路由原理

* + 1. document路由到shard上是什么意思？
    2. 路由算法：

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

举个例子，一个index有3个primary shard，P0，P1，P2

每次增删改查一个document的时候，都会带过来一个routing number，默认就是这个document的\_id（可能是手动指定，也可能是自动生成）

routing = \_id，假设\_id=1

会将这个routing值，传入一个hash函数中，产出一个routing值的hash值，hash(routing) = 21

然后将hash函数产出的值对这个index的primary shard的数量求余数，21 % 3 = 0

就决定了，这个document就放在P0上。

决定一个document在哪个shard上，最重要的一个值就是routing值，默认是\_id，也可以手动指定，相同的routing值，每次过来，从hash函数中，产出的hash值一定是相同的

无论hash值是几，无论是什么数字，对number\_of\_primary\_shards求余数，结果一定是在0~number\_of\_primary\_shards-1之间这个范围内的。0,1,2。

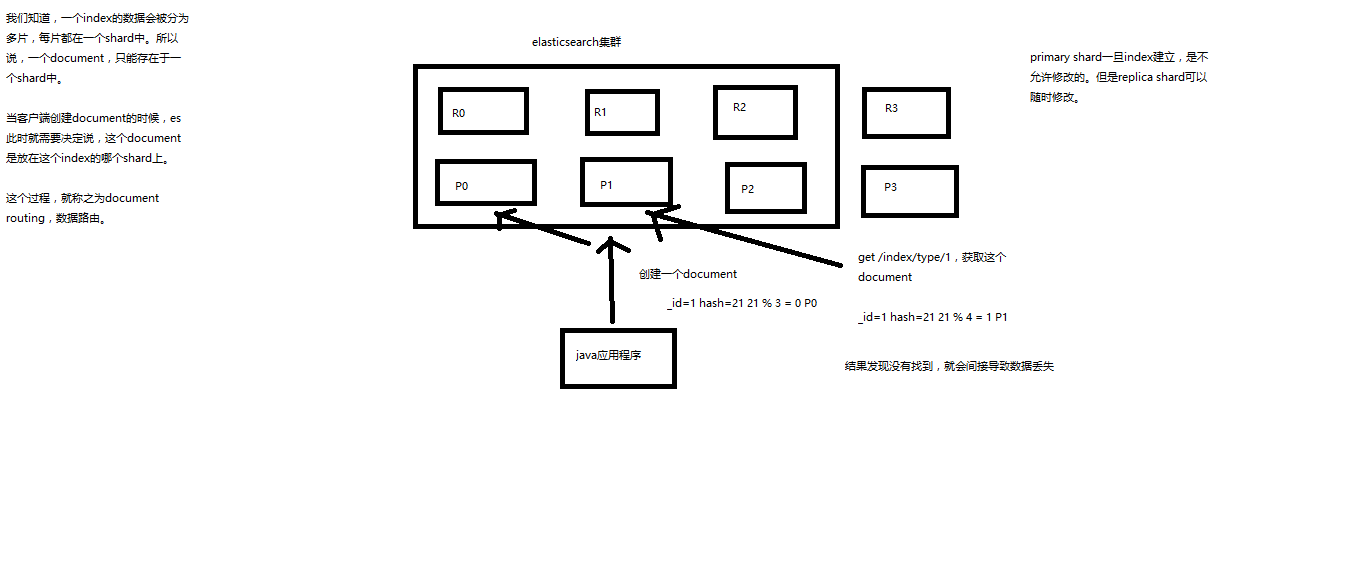
* + 1. \_id or custom routing value

默认的routing就是\_id

也可以在发送请求的时候，手动指定一个routing value，比如说put /index/type/id?routing=user\_id

手动指定routing value是很有用的，可以保证说，某一类document一定被路由到一个shard上去，那么在后续进行应用级别的负载均衡，以及提升批量读取的性能的时候，是很有帮助的

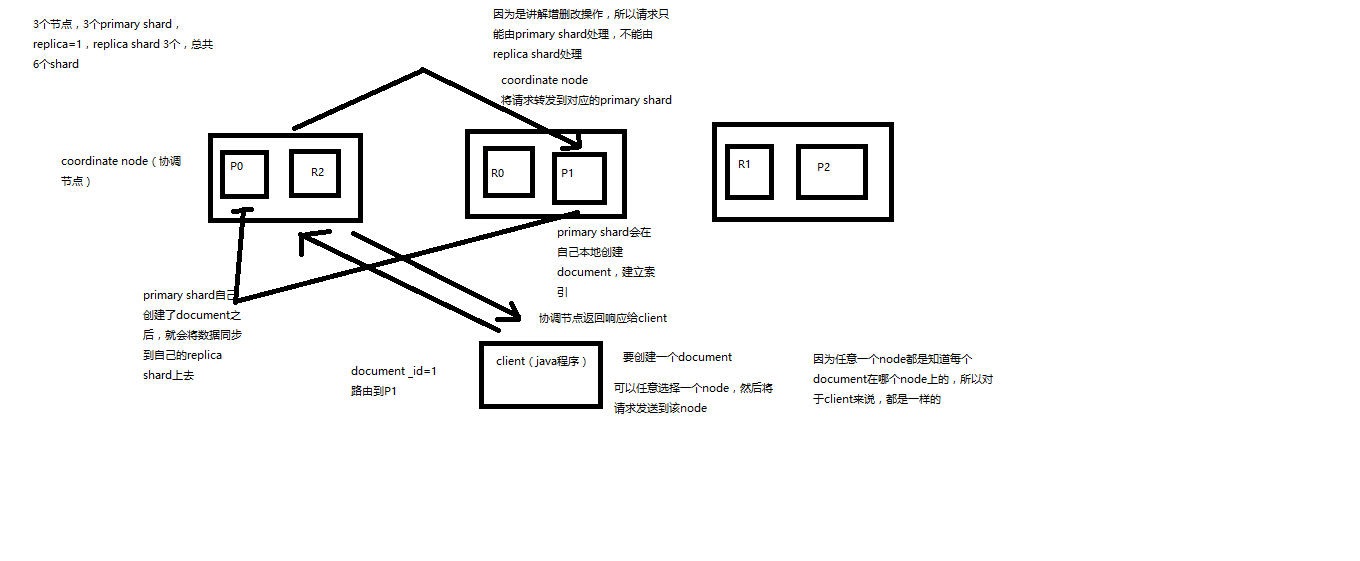
* + 1. primary shard数量不可变的谜底



**document路由原理**

## document增删改内部原理

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（有primary shard）
3. 实际的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replica node都搞定之后，就返回响应结果给客户端



**es增删改内部原理**

## 写一致性原理以及quorum机制

1. consistency，one（primary shard），all（all shard），quorum（default）

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

1. 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

1. 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

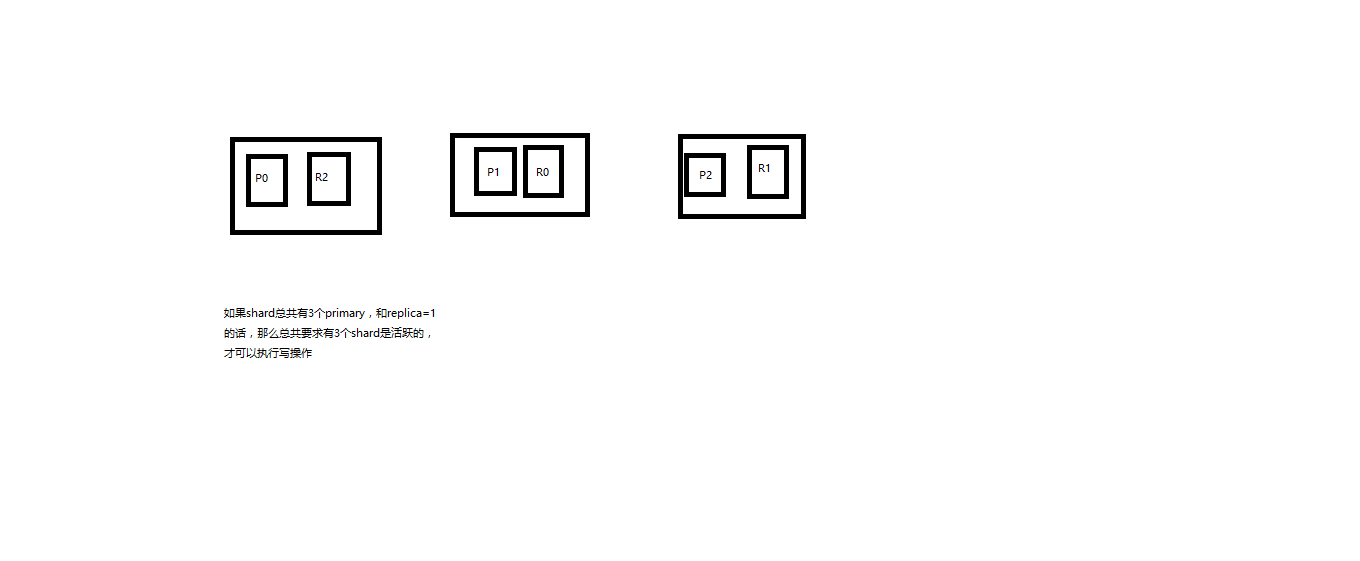
es提供了一种特殊的处理场景，就是说当number\_of\_replicas>1时才生效，因为假如说，你就一个primary shard，replica=1，此时就2个shard

(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

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

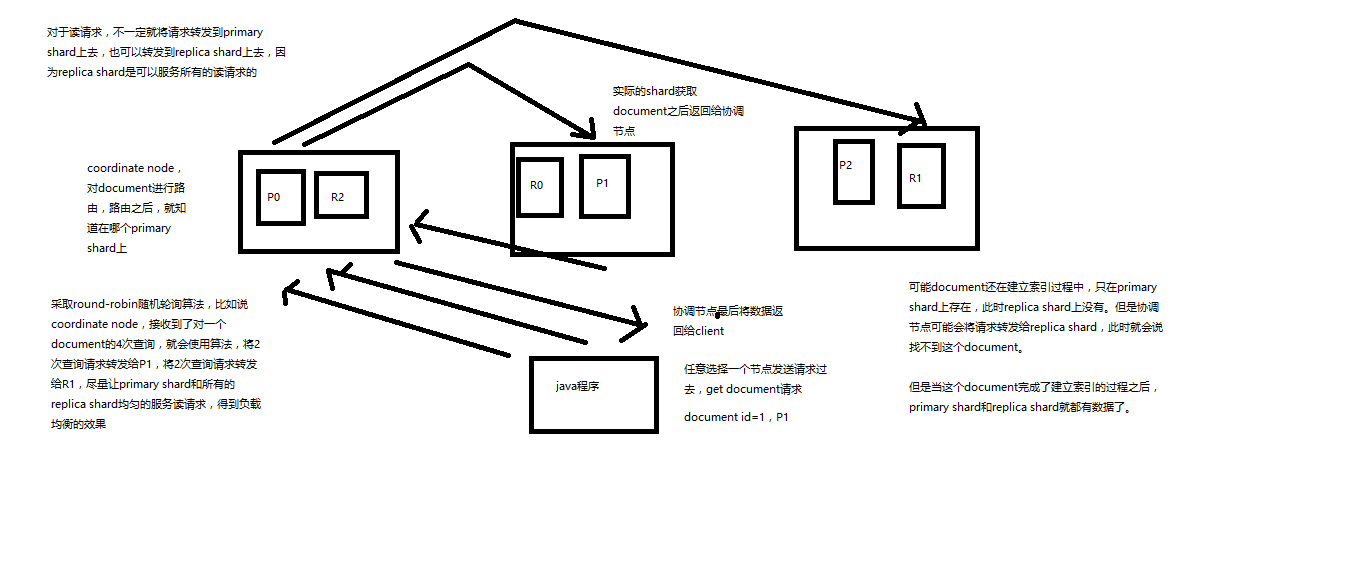
等待期间，期望活跃的shard数量可以增加，最后实在不行，就会timeout

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

**写一致性**

## document查询内部原理

1. 客户端发送请求到任意一个node，成为coordinate node
2. coordinate node对document进行路由，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
5. 特殊情况：document如果还在建立索引过程中，可能只有primary shard有，任何一个replica shard都没有，此时可能会导致无法读取到document，但是document完成索引建立之后，primary shard和replica shard就都有了

****

**读请求内部原理**

## bulk api的奇特json格式与底层性能优化关系

bulk api奇特的json格式

{"action": {"meta"}}\n

{"data"}\n

{"action": {"meta"}}\n

{"data"}\n

[{

"action": {

},

"data": {

}

}]

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

{"action": {"meta"}}\n

{"data"}\n

{"action": {"meta"}}\n

{"data"}\n

* 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

## search结果（search timeout机制）

* + 1. 我们如果发出一个搜索请求的话，会拿到一堆搜索结果，这个搜索结果里的各种数据，都代表了什么含义

GET /\_search

{

"took": 6,

"timed\_out": false,

"\_shards": {

"total": 6,

"successful": 6,

"failed": 0

},

"hits": {

"total": 10,

"max\_score": 1,

"hits": [

{

"\_index": ".kibana",

"\_type": "config",

"\_id": "5.2.0",

"\_score": 1,

"\_source": {

"buildNum": 14695

}

}

]

}

}

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

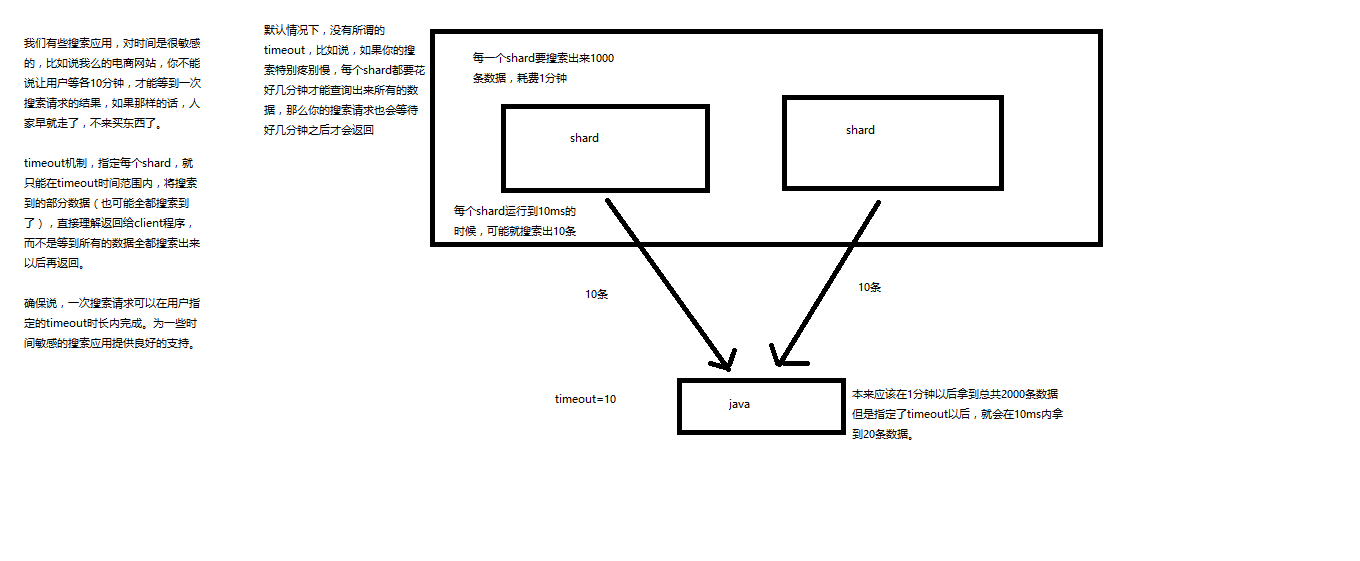
shards：shards fail的条件（primary和replica全部挂掉），不影响其他shard。默认情况下来说，一个搜索请求，会打到一个index的所有primary shard上去，当然了，每个primary shard都可能会有一个或多个replic shard，所以请求也可以到primary shard的其中一个replica shard上去。

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

* + 1. 搜索的timeout机制，底层的原理，画图讲解

**timeout机制详解**

## multi-index和multi-type搜索模式及其原理

* + 1. multi-index和multi-type搜索模式

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

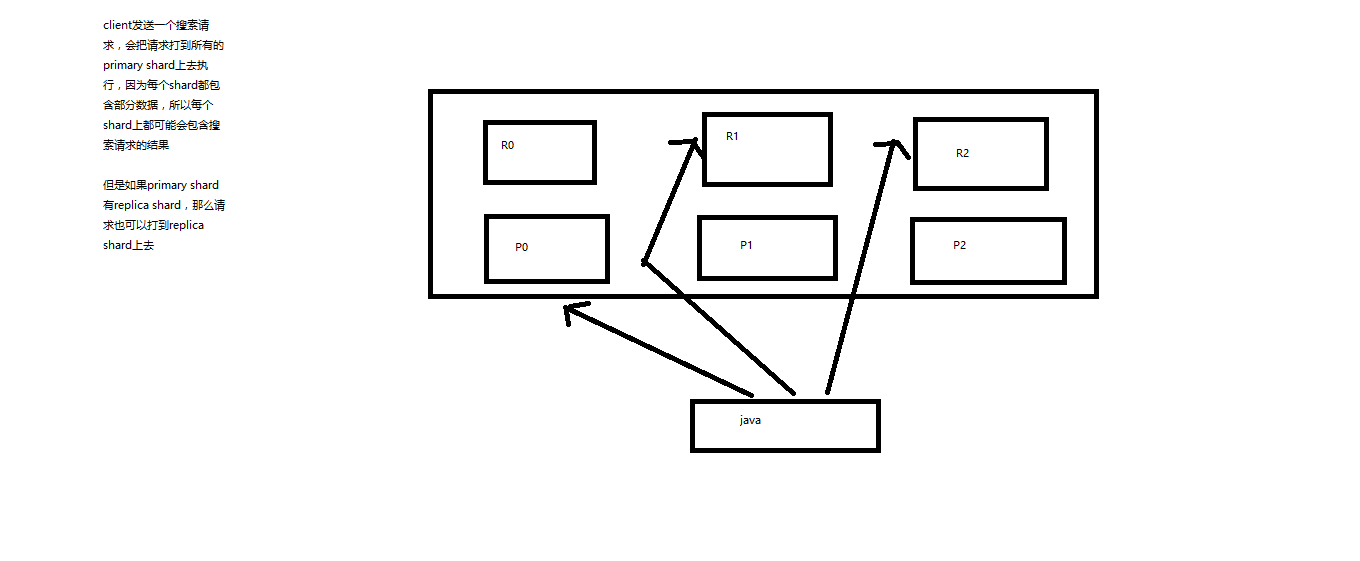
/index1/type1,type2/\_search：可以搜索一个index下多个type的数据

/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

* + 1. 搜索原理

搜索原理初步图解



**初步图解搜索原理**

## es进行分页搜索的语法

* + 1. 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

实验：

GET /test\_index/test\_type/\_search

"hits": {

"total": 9,

"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

{

"took": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 9,

"max\_score": 1,

"hits": [

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_score": 1,

"\_source": {

"test\_field": "test client 2"

}

},

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "6",

"\_score": 1,

"\_source": {

"test\_field": "tes test"

}

},

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "4",

"\_score": 1,

"\_source": {

"test\_field": "test4"

}

}

]

}

}

第一页：id=8,6,4

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

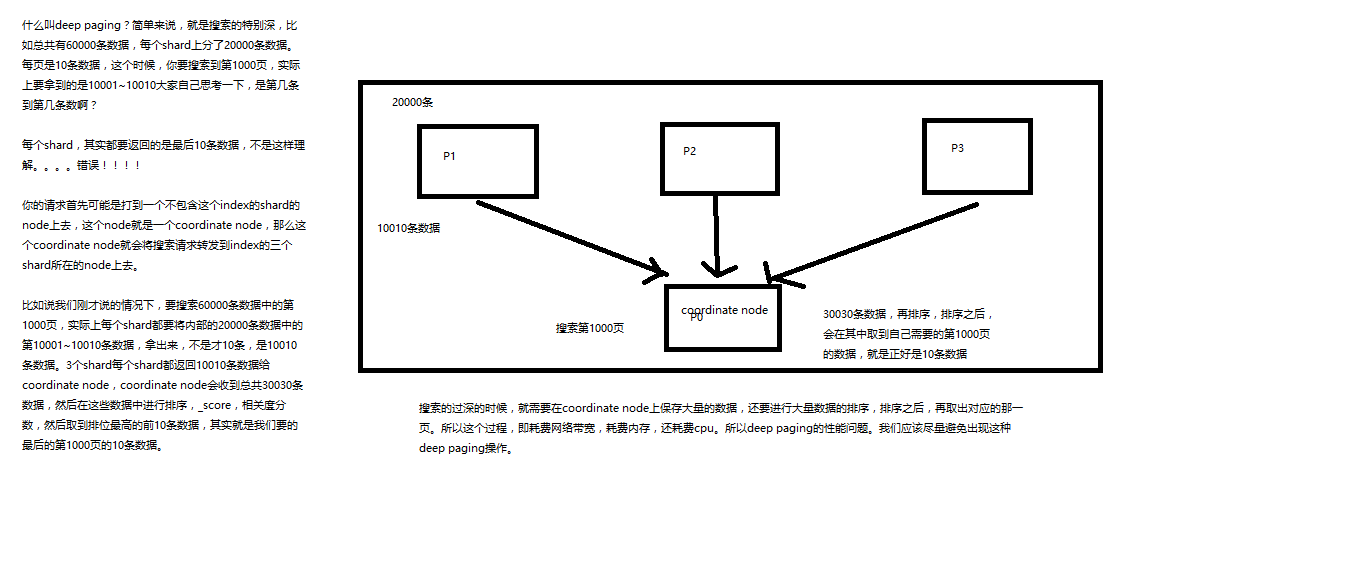
第二页：id=2,自动生成,7

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

第三页：id=1,11,3

* + 1. 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？

deep paging性能问题，以及原理深度图解揭秘，很高级的知识点

**deep paging图解**

## query string search语法以及\_all metadata原理

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

1. \_all metadata的原理和作用

GET /test\_index/test\_type/\_search?q=test

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

{

"name": "jack",

"age": 26,

"email": "jack@sina.com",

"address": "guamgzhou"

}

"jack 26 jack@sina.com 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到底是什么

插入几条数据，让es自动为我们建立一个索引

PUT /website/article/1

{

"post\_date": "2017-01-01",

"title": "my first article",

"content": "this is my first article in this website",

"author\_id": 11400

}

PUT /website/article/2

{

"post\_date": "2017-01-02",

"title": "my second article",

"content": "this is my second article in this website",

"author\_id": 11400

}

PUT /website/article/3

{

"post\_date": "2017-01-03",

"title": "my third article",

"content": "this is my third article in this website",

"author\_id": 11400

}

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

{

"website": {

"mappings": {

"article": {

"properties": {

"author\_id": {

"type": "long"

},

"content": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"post\_date": {

"type": "date"

},

"title": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

}

}

}

}

Text：会分词，然后进行索引

支持模糊、精确查询

不支持聚合

keyword：不进行分词，直接索引

支持模糊、精确查询

支持聚合

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。

## 精确匹配与全文搜索的对比分析

1. exact value

2017-01-01，exact value，搜索的时候，必须输入2017-01-01，才能搜索出来

如果你输入一个01，是搜索不出来的

1. full text

（1）缩写 vs. 全程：cn vs. china

（2）格式转化：like liked likes

（3）大小写：Tom vs tom

（4）同义词：like vs love

2017-01-01，2017 01 01，搜索2017，或者01，都可以搜索出来

china，搜索cn，也可以将china搜索出来

likes，搜索like，也可以将likes搜索出来

Tom，搜索tom，也可以将Tom搜索出来

like，搜索love，同义词，也可以将like搜索出来

就不是说单纯的只是匹配完整的一个值，而是可以对值进行拆分词语后（分词）进行匹配，也可以通过缩写、时态、大小写、同义词等进行匹配

## 倒排索引核心原理

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 \*

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.

## 分词器的内部组成到底是什么，以及内置分词器的介绍

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），分词器

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（<span>hello<span> --> hello），& --> and（I&you --> I 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

1. 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

## query string的分词以及mapping遗留问题

1、query string分词

query string必须以和index建立时相同的analyzer进行分词

query string对exact value和full text的区别对待

date：exact value

\_all：full text

比如我们有一个document，其中有一个field，包含的value是：hello you and me，建立倒排索引

我们要搜索这个document对应的index，搜索文本是hell me，这个搜索文本就是query string

query string，默认情况下，es会使用它对应的field建立倒排索引时相同的分词器去进行分词，分词和normalization，只有这样，才能实现正确的搜索

我们建立倒排索引的时候，将dogs --> dog，结果你搜索的时候，还是一个dogs，那不就搜索不到了吗？所以搜索的时候，那个dogs也必须变成dog才行。才能搜索到。

知识点：不同类型的field，可能有的就是full text，有的就是exact value

post\_date，date：exact value

\_all：full text，分词，normalization

2、mapping引入案例遗留问题大揭秘

GET /\_search?q=2017

搜索的是\_all field，document所有的field都会拼接成一个大串，进行分词

2017-01-02 my second article this is my second article in this website 11400

doc1 doc2 doc3

2017 \* \* \*

01 \*

02 \*

03 \*

\_all，2017，自然会搜索到3个docuemnt

GET /\_search?q=2017-01-01

\_all，2017-01-01，query string会用跟建立倒排索引一样的分词器去进行分词

2017

01

01

GET /\_search?q=post\_date:2017-01-01

date，会作为exact value去建立索引

doc1 doc2 doc3

2017-01-01 \*

2017-01-02 \*

2017-01-03 \*

post\_date:2017-01-01，2017-01-01，doc1一条document

GET /\_search?q=post\_date:2017，这个在这里不讲解，因为是es 5.2以后做的一个优化

3、测试分词器

GET /\_analyze

{

"analyzer": "standard",

"text": "Text to analyze"

}

## 什么是mapping透彻理解

* 1. 往es里面直接插入数据，es会自动建立索引，同时建立type以及对应的mapping
  2. mapping中就自动定义了每个field的数据类型
  3. 不同的数据类型（比如说text和date），可能有的是exact value，有的是full text
  4. exact value，在建立倒排索引的时候，分词的时候，是将整个值一起作为一个关键词建立到倒排索引中的；full text，会经历各种各样的处理，分词，normaliztion（时态转换，同义词转换，大小写转换），才会建立到倒排索引中
  5. 同时呢，exact value和full text类型的field就决定了，在一个搜索过来的时候，对exact value field或者是full text field进行搜索的行为也是不一样的，会跟建立倒排索引的行为保持一致；比如说exact value搜索的时候，就是直接按照整个值进行匹配，full text query string，也会进行分词和normalization再去倒排索引中去搜索
  6. 可以用es的dynamic mapping，让其自动建立mapping，包括自动设置数据类型；也可以提前手动创建index和type的mapping，自己对各个field进行设置，包括数据类型，包括索引行为，包括分词器，等等

mapping，就是index的type的元数据，每个type都有一个自己的mapping，决定了数据类型，建立倒排索引的行为，还有进行搜索的行为

## \_mapping的核心数据类型以及dynamic mapping

1、核心的数据类型

string

byte，short，integer，long

float，double

boolean

date

2、dynamic mapping

true or false --> boolean

123 --> long

123.45 --> double

2017-01-01 --> date

"hello world" --> string/text

3、查看mapping

GET /index/\_mapping/type

## 手动建立和修改mapping以及定制string类型数据是否分词

1、如何建立索引

analyzed: 根据分词器分词，可以根据原词和分词后的词条索引Field

not\_analyzed: 不会分词，只能根据原词索引Field

no : 该字段不会被索引，查不到

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

PUT /website

{

"mappings": {

"article": {

"properties": {

"author\_id": {

"type": "long"

},

"title": {

"type": "text",

"analyzer": "english"

},

"content": {

"type": "text"

},

"post\_date": {

"type": "date"

},

"publisher\_id": {

"type": "text",

"index": "not\_analyzed"

}

}

}

}

}

PUT /website

{

"mappings": {

"article": {

"properties": {

"author\_id": {

"type": "text"

}

}

}

}

}

{

"error": {

"root\_cause": [

{

"type": "index\_already\_exists\_exception",

"reason": "index [website/co1dgJ-uTYGBEEOOL8GsQQ] already exists",

"index\_uuid": "co1dgJ-uTYGBEEOOL8GsQQ",

"index": "website"

}

],

"type": "index\_already\_exists\_exception",

"reason": "index [website/co1dgJ-uTYGBEEOOL8GsQQ] already exists",

"index\_uuid": "co1dgJ-uTYGBEEOOL8GsQQ",

"index": "website"

},

"status": 400

}

PUT /website/\_mapping/article

{

"properties" : {

"new\_field" : {

"type" : "string",

"index": "not\_analyzed"

}

}

}

3、测试mapping

GET /website/\_analyze

{

"field": "content",

"text": "my-dogs"

}

GET website/\_analyze

{

"field": "new\_field",

"text": "my dogs"

}

{

"error": {

"root\_cause": [

{

"type": "remote\_transport\_exception",

"reason": "[4onsTYV][127.0.0.1:9300][indices:admin/analyze[s]]"

}

],

"type": "illegal\_argument\_exception",

"reason": "Can't process field [new\_field], Analysis requests are only supported on tokenized fields"

},

"status": 400

}

## mapping复杂数据类型以及object类型数据底层结构

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

null，[]，[null]

3、object field

PUT /company/employee/1

{

"address": {

"country": "china",

"province": "guangdong",

"city": "guangzhou"

},

"name": "jack",

"age": 27,

"join\_date": "2017-01-01"

}

address：object类型

{

"company": {

"mappings": {

"employee": {

"properties": {

"address": {

"properties": {

"city": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"country": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"province": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

},

"age": {

"type": "long"

},

"join\_date": {

"type": "date"

},

"name": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

}

}

}

}

{

"address": {

"country": "china",

"province": "guangdong",

"city": "guangzhou"

},

"name": "jack",

"age": 27,

"join\_date": "2017-01-01"

}

{

"name": [jack],

"age": [27],

"join\_date": [2017-01-01],

"address.country": [china],

"address.province": [guangdong],

"address.city": [guangzhou]

}

{

"authors": [

{ "age": 26, "name": "Jack White"},

{ "age": 55, "name": "Tom Jones"},

{ "age": 39, "name": "Kitty Smith"}

]

}

{

"authors.age": [26, 55, 39],

"authors.name": [jack, white, tom, jones, kitty, smith]

}

## search api的基础语法介绍

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

## Query DSL搜索语法

1、一个例子让你明白什么是Query DSL

GET /\_search

{

"query": {

"match\_all": {}

}

}

2、Query DSL的基本语法

{

QUERY\_NAME: {

ARGUMENT: VALUE,

ARGUMENT: VALUE,...

}

}

{

QUERY\_NAME: {

FIELD\_NAME: {

ARGUMENT: VALUE,

ARGUMENT: VALUE,...

}

}

}

示例：

GET /test\_index/test\_type/\_search

{

"query": {

"match": {

"test\_field": "test"

}

}

}

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

{

"took": 1,

"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": "my hadoop article",

"content": "hadoop is very bad",

"author\_id": 111

}

},

{

"\_index": "website",

"\_type": "article",

"\_id": "1",

"\_score": 1,

"\_source": {

"title": "my elasticsearch article",

"content": "es is very bad",

"author\_id": 110

}

},

{

"\_index": "website",

"\_type": "article",

"\_id": "3",

"\_score": 1,

"\_source": {

"title": "my elasticsearch article",

"content": "es is very goods",

"author\_id": 111

}

}

]

}

}

GET /website/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"title": "elasticsearch"

}

}

],

"should": [

{

"match": {

"content": "elasticsearch"

}

}

],

"must\_not": [

{

"match": {

"author\_id": 111

}

}

]

}

}

}

GET /test\_index/\_search

{

"query": {

"bool": {

"must": { "match": { "name": "tom" }},

"should": [

{ "match": { "hired": true }},

{ "bool": {

"must": { "match": { "personality": "good" }},

"must\_not": { "match": { "rude": true }}

}}

],

"minimum\_should\_match": 1

}

}

}

## filter与query对比：相关度，性能

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

## 常用的各种query搜索语法

1、match all

GET /\_search

{

"query": {

"match\_all": {}

}

}

2、match

GET /\_search

{

"query": { "match": { "title": "my elasticsearch article" }}

}

3、multi match

GET /test\_index/test\_type/\_search

{

"query": {

"multi\_match": {

"query": "test",

"fields": ["test\_field", "test\_field1"]

}

}

}

4、range query

GET /company/employee/\_search

{

"query": {

"range": {

"age": {

"gte": 30

}

}

}

}

5、term query

GET /test\_index/test\_type/\_search

{

"query": {

"term": {

"test\_field": "test hello"

}

}

}

6、terms query

GET /\_search

{

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

}

7、exist query（2.x中的查询，现在已经不提供了）

## 多搜索条件组合查询

GET /website/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"title": "elasticsearch"

}

}

],

"should": [

{

"match": {

"content": "elasticsearch"

}

}

],

"must\_not": [

{

"match": {

"author\_id": 111

}

}

]

}

}

}

{

"bool": {

"must": { "match": { "title": "how to make millions" }},

"must\_not": { "match": { "tag": "spam" }},

"should": [

{ "match": { "tag": "starred" }}

],

"filter": {

"range": { "date": { "gte": "2014-01-01" }}

}

}

}

bool

must，must\_not，should，filter

每个子查询都会计算一个document针对它的相关度分数，然后bool综合所有分数，合并为一个分数，当然filter是不会计算分数的

{

"bool": {

"must": { "match": { "title": "how to make millions" }},

"must\_not": { "match": { "tag": "spam" }},

"should": [

{ "match": { "tag": "starred" }}

],

"filter": {

"bool": {

"must": [

{ "range": { "date": { "gte": "2014-01-01" }}},

{ "range": { "price": { "lte": 29.99 }}}

],

"must\_not": [

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

]

}

}

}

}

GET /company/employee/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"age": {

"gte": 30

}

}

}

}

}

}

## 如何定位不合法的搜索以及其原因

GET /test\_index/test\_type/\_validate/query?explain

{

"query": {

"math": {

"test\_field": "test"

}

}

}

{

"valid": false,

"error": "org.elasticsearch.common.ParsingException: no [query] registered for [math]"

}

GET /test\_index/test\_type/\_validate/query?explain

{

"query": {

"match": {

"test\_field": "test"

}

}

}

{

"valid": true,

"\_shards": {

"total": 1,

"successful": 1,

"failed": 0

},

"explanations": [

{

"index": "test\_index",

"valid": true,

"explanation": "+test\_field:test #(#\_type:test\_type)"

}

]

}

一般用在那种特别复杂庞大的搜索下，比如你一下子写了上百行的搜索，这个时候可以先用validate api去验证一下，搜索是否合法

## 如何定制搜索结果的排序规则

1、默认排序规则

默认情况下，是按照\_score降序排序的

然而，某些情况下，可能没有有用的\_score，比如说filter

GET /\_search

{

"query" : {

"bool" : {

"filter" : {

"term" : {

"author\_id" : 1

}

}

}

}

}

当然，也可以是constant\_score

GET /\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"author\_id" : 1

}

}

}

}

}

2、定制排序规则

GET /company/employee/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"age": {

"gte": 30

}

}

}

}

},

"sort": [

{

"join\_date": {

"order": "asc"

}

}

]

}

## 如何将一个field索引两次来解决字符串排序问题

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

fielddata=true: 将正排索引数据加载到内存中, 可以对分词的field执行聚合操作，而且会消耗很大的内存

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"

}

}

]

}

## 相关度评分TF&IDF算法

1、算法介绍

relevance score算法，简单来说，就是计算出，一个索引中的文本，与搜索文本，他们之间的关联匹配程度

Elasticsearch使用的是 term frequency/inverse document frequency算法，简称为TF/IDF算法

Term frequency：搜索文本中的各个词条在field文本中出现了多少次，出现次数越多，就越相关

搜索请求：hello world

doc1：hello you, and world is very good

doc2：hello, how are you

Inverse document frequency：搜索文本中的各个词条在整个索引的所有文档中出现了多少次，出现的次数越多，就越不相关

搜索请求：hello world

doc1：hello, today is very good

doc2：hi world, how are you

比如说，在index中有1万条document，hello这个单词在所有的document中，一共出现了1000次；world这个单词在所有的document中，一共出现了100次

doc2更相关

Field-length norm：field长度，field越长，相关度越弱

搜索请求：hello world

doc1：{ "title": "hello article", "content": "babaaba 1万个单词" }

doc2：{ "title": "my article", "content": "blablabala 1万个单词，hi world" }

hello world在整个index中出现的次数是一样多的

doc1更相关，title field更短

2、score是如何被计算出来的

GET /test\_index/test\_type/\_search?explain

{

"query": {

"match": {

"test\_field": "test hello"

}

}

}

{

"took": 6,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 4,

"max\_score": 1.595089,

"hits": [

{

"\_shard": "[test\_index][2]",

"\_node": "4onsTYVZTjGvIj9\_spWz2w",

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "20",

"\_score": 1.595089,

"\_source": {

"test\_field": "test hello"

},

"\_explanation": {

"value": 1.595089,

"description": "sum of:",

"details": [

{

"value": 1.595089,

"description": "sum of:",

"details": [

{

"value": 0.58279467,

"description": "weight(test\_field:test in 0) [PerFieldSimilarity], result of:",

"details": [

{

"value": 0.58279467,

"description": "score(doc=0,freq=1.0 = termFreq=1.0\n), product of:",

"details": [

{

"value": 0.6931472,

"description": "idf, computed as log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",

"details": [

{

"value": 2,

"description": "docFreq",

"details": []

},

{

"value": 4,

"description": "docCount",

"details": []

}

]

},

{

"value": 0.840795,

"description": "tfNorm, computed as (freq \* (k1 + 1)) / (freq + k1 \* (1 - b + b \* fieldLength / avgFieldLength)) from:",

"details": [

{

"value": 1,

"description": "termFreq=1.0",

"details": []

},

{

"value": 1.2,

"description": "parameter k1",

"details": []

},

{

"value": 0.75,

"description": "parameter b",

"details": []

},

{

"value": 1.75,

"description": "avgFieldLength",

"details": []

},

{

"value": 2.56,

"description": "fieldLength",

"details": []

}

]

}

]

}

]

},

{

"value": 1.0122943,

"description": "weight(test\_field:hello in 0) [PerFieldSimilarity], result of:",

"details": [

{

"value": 1.0122943,

"description": "score(doc=0,freq=1.0 = termFreq=1.0\n), product of:",

"details": [

{

"value": 1.2039728,

"description": "idf, computed as log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",

"details": [

{

"value": 1,

"description": "docFreq",

"details": []

},

{

"value": 4,

"description": "docCount",

"details": []

}

]

},

{

"value": 0.840795,

"description": "tfNorm, computed as (freq \* (k1 + 1)) / (freq + k1 \* (1 - b + b \* fieldLength / avgFieldLength)) from:",

"details": [

{

"value": 1,

"description": "termFreq=1.0",

"details": []

},

{

"value": 1.2,

"description": "parameter k1",

"details": []

},

{

"value": 0.75,

"description": "parameter b",

"details": []

},

{

"value": 1.75,

"description": "avgFieldLength",

"details": []

},

{

"value": 2.56,

"description": "fieldLength",

"details": []

}

]

}

]

}

]

}

]

},

{

"value": 0,

"description": "match on required clause, product of:",

"details": [

{

"value": 0,

"description": "# clause",

"details": []

},

{

"value": 1,

"description": "\*:\*, product of:",

"details": [

{

"value": 1,

"description": "boost",

"details": []

},

{

"value": 1,

"description": "queryNorm",

"details": []

}

]

}

]

}

]

}

},

{

"\_shard": "[test\_index][2]",

"\_node": "4onsTYVZTjGvIj9\_spWz2w",

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "6",

"\_score": 0.58279467,

"\_source": {

"test\_field": "tes test"

},

"\_explanation": {

"value": 0.58279467,

"description": "sum of:",

"details": [

{

"value": 0.58279467,

"description": "sum of:",

"details": [

{

"value": 0.58279467,

"description": "weight(test\_field:test in 0) [PerFieldSimilarity], result of:",

"details": [

{

"value": 0.58279467,

"description": "score(doc=0,freq=1.0 = termFreq=1.0\n), product of:",

"details": [

{

"value": 0.6931472,

"description": "idf, computed as log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",

"details": [

{

"value": 2,

"description": "docFreq",

"details": []

},

{

"value": 4,

"description": "docCount",

"details": []

}

]

},

{

"value": 0.840795,

"description": "tfNorm, computed as (freq \* (k1 + 1)) / (freq + k1 \* (1 - b + b \* fieldLength / avgFieldLength)) from:",

"details": [

{

"value": 1,

"description": "termFreq=1.0",

"details": []

},

{

"value": 1.2,

"description": "parameter k1",

"details": []

},

{

"value": 0.75,

"description": "parameter b",

"details": []

},

{

"value": 1.75,

"description": "avgFieldLength",

"details": []

},

{

"value": 2.56,

"description": "fieldLength",

"details": []

}

]

}

]

}

]

}

]

},

{

"value": 0,

"description": "match on required clause, product of:",

"details": [

{

"value": 0,

"description": "# clause",

"details": []

},

{

"value": 1,

"description": "\*:\*, product of:",

"details": [

{

"value": 1,

"description": "boost",

"details": []

},

{

"value": 1,

"description": "queryNorm",

"details": []

}

]

}

]

}

]

}

},

{

"\_shard": "[test\_index][3]",

"\_node": "4onsTYVZTjGvIj9\_spWz2w",

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "7",

"\_score": 0.5565415,

"\_source": {

"test\_field": "test client 2"

},

"\_explanation": {

"value": 0.5565415,

"description": "sum of:",

"details": [

{

"value": 0.5565415,

"description": "sum of:",

"details": [

{

"value": 0.5565415,

"description": "weight(test\_field:test in 0) [PerFieldSimilarity], result of:",

"details": [

{

"value": 0.5565415,

"description": "score(doc=0,freq=1.0 = termFreq=1.0\n), product of:",

"details": [

{

"value": 0.6931472,

"description": "idf, computed as log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",

"details": [

{

"value": 1,

"description": "docFreq",

"details": []

},

{

"value": 2,

"description": "docCount",

"details": []

}

]

},

{

"value": 0.8029196,

"description": "tfNorm, computed as (freq \* (k1 + 1)) / (freq + k1 \* (1 - b + b \* fieldLength / avgFieldLength)) from:",

"details": [

{

"value": 1,

"description": "termFreq=1.0",

"details": []

},

{

"value": 1.2,

"description": "parameter k1",

"details": []

},

{

"value": 0.75,

"description": "parameter b",

"details": []

},

{

"value": 2.5,

"description": "avgFieldLength",

"details": []

},

{

"value": 4,

"description": "fieldLength",

"details": []

}

]

}

]

}

]

}

]

},

{

"value": 0,

"description": "match on required clause, product of:",

"details": [

{

"value": 0,

"description": "# clause",

"details": []

},

{

"value": 1,

"description": "\_type:test\_type, product of:",

"details": [

{

"value": 1,

"description": "boost",

"details": []

},

{

"value": 1,

"description": "queryNorm",

"details": []

}

]

}

]

}

]

}

},

{

"\_shard": "[test\_index][1]",

"\_node": "4onsTYVZTjGvIj9\_spWz2w",

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_score": 0.25316024,

"\_source": {

"test\_field": "test client 2"

},

"\_explanation": {

"value": 0.25316024,

"description": "sum of:",

"details": [

{

"value": 0.25316024,

"description": "sum of:",

"details": [

{

"value": 0.25316024,

"description": "weight(test\_field:test in 0) [PerFieldSimilarity], result of:",

"details": [

{

"value": 0.25316024,

"description": "score(doc=0,freq=1.0 = termFreq=1.0\n), product of:",

"details": [

{

"value": 0.2876821,

"description": "idf, computed as log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",

"details": [

{

"value": 1,

"description": "docFreq",

"details": []

},

{

"value": 1,

"description": "docCount",

"details": []

}

]

},

{

"value": 0.88,

"description": "tfNorm, computed as (freq \* (k1 + 1)) / (freq + k1 \* (1 - b + b \* fieldLength / avgFieldLength)) from:",

"details": [

{

"value": 1,

"description": "termFreq=1.0",

"details": []

},

{

"value": 1.2,

"description": "parameter k1",

"details": []

},

{

"value": 0.75,

"description": "parameter b",

"details": []

},

{

"value": 3,

"description": "avgFieldLength",

"details": []

},

{

"value": 4,

"description": "fieldLength",

"details": []

}

]

}

]

}

]

}

]

},

{

"value": 0,

"description": "match on required clause, product of:",

"details": [

{

"value": 0,

"description": "# clause",

"details": []

},

{

"value": 1,

"description": "\*:\*, product of:",

"details": [

{

"value": 1,

"description": "boost",

"details": []

},

{

"value": 1,

"description": "queryNorm",

"details": []

}

]

}

]

}

]

}

}

]

}

}

3、分析一个document是如何被匹配上的

GET /test\_index/test\_type/6/\_explain

{

"query": {

"match": {

"test\_field": "test hello"

}

}

}

## 内核级知识点之doc value

搜索的时候，要依靠倒排索引；排序的时候，需要依靠正排索引，看到每个document的每个field，然后进行排序，所谓的正排索引，其实就是doc values

在建立索引的时候，一方面会建立倒排索引，以供搜索用；一方面会建立正排索引，也就是doc values，以供排序，聚合，过滤等操作使用

doc values是被保存在磁盘上的，此时如果内存足够，os会自动将其缓存在内存中，性能还是会很高；如果内存不足够，os会将其写入磁盘上

doc1: hello world you and me

doc2: hi, world, how are you

word doc1 doc2

hello \*

world \* \*

you \* \*

and \*

me \*

hi \*

how \*

are \*

hello you --> hello, you

hello --> doc1

you --> doc1,doc2

doc1: hello world you and me

doc2: hi, world, how are you

sort by age

doc1: { "name": "jack", "age": 27 }

doc2: { "name": "tom", "age": 30 }

document name age

doc1 jack 27

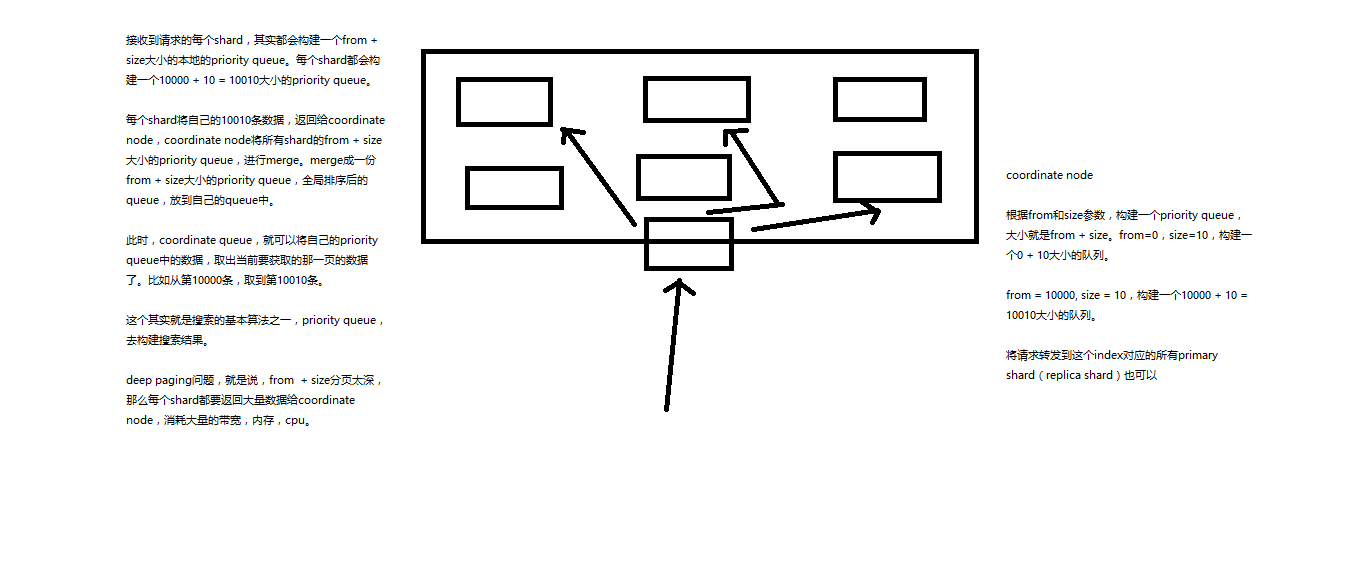
doc2 tom 30

## query phase

1、query phase

* 1. 搜索请求发送到某一个coordinate node，构构建一个priority queue，长度以paging操作from和size为准，默认为10
  2. coordinate node将请求转发到所有shard，每个shard本地搜索，并构建一个本地的priority queue
  3. 各个shard将自己的priority queue返回给coordinate node，并构建一个全局的priority queue

2、replica shard如何提升搜索吞吐量

一次请求要打到所有shard的一个replica/primary上去，如果每个shard都有多个replica，那么同时并发过来的搜索请求可以同时打到其他的replica上去

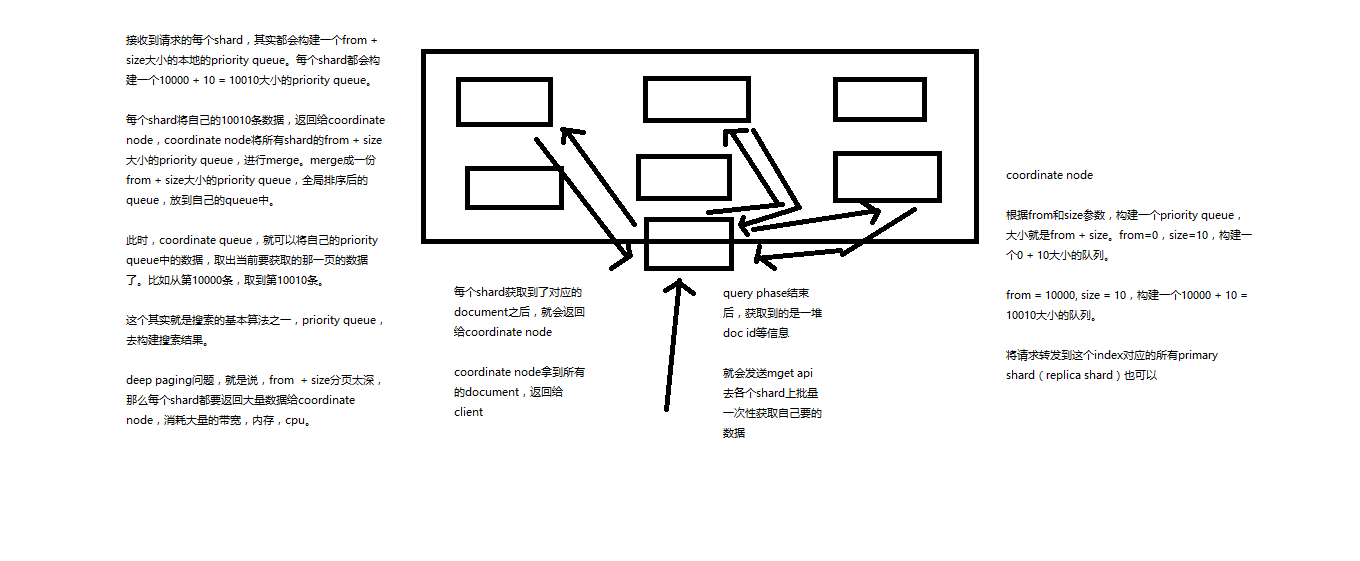
**query phase**

## fetch phase

1、fetch phbase工作流程

* 1. coordinate node构建完priority queue之后，就发送mget请求去所有shard上获取对应的document
  2. 各个shard将document返回给coordinate node
  3. coordinate node将合并后的document结果返回给client客户端

2、一般搜索，如果不加from和size，就默认搜索前10条，按照\_score排序

**fetch phase**

## 搜索相关参数梳理以及bouncing results问题解决方案

1、preference

决定了哪些shard会被用来执行搜索操作

\_primary, \_primary\_first, \_local, \_only\_node:xyz, \_prefer\_node:xyz, \_shards:2,3

bouncing results问题，两个document排序，field值相同；不同的shard上，可能排序不同；每次请求轮询打到不同的replica shard上；每次页面上看到的搜索结果的排序都不一样。这就是bouncing result，也就是跳跃的结果。

搜索的时候，是轮询将搜索请求发送到每一个replica shard（primary shard），但是在不同的shard上，可能document的排序不同

解决方案就是将preference设置为一个字符串，比如说user\_id，让每个user每次搜索的时候，都使用同一个replica shard去执行，就不会看到bouncing results了

2、timeout，已经讲解过原理了，主要就是限定在一定时间内，将部分获取到的数据直接返回，避免查询耗时过长

3、routing，document文档路由，\_id路由，routing=user\_id，这样的话可以让同一个user对应的数据到一个shard上去

4、search\_type

default：query\_then\_fetch

dfs\_query\_then\_fetch，可以提升revelance sort精准度

## 基于scoll技术滚动搜索大量数据

如果一次性要查出来比如10万条数据，那么性能会很差，此时一般会采取用scoll滚动查询，一批一批的查，直到所有数据都查询完处理完

使用scoll滚动搜索，可以先搜索一批数据，然后下次再搜索一批数据，以此类推，直到搜索出全部的数据来

scoll搜索会在第一次搜索的时候，保存一个当时的视图快照，之后只会基于该旧的视图快照提供数据搜索，如果这个期间数据变更，是不会让用户看到的

采用基于\_doc进行排序的方式，性能较高

每次发送scroll请求，我们还需要指定一个scoll参数，指定一个时间窗口，每次搜索请求只要在这个时间窗口内能完成就可以了

GET /test\_index/test\_type/\_search?scroll=1m

{

"query": {

"match\_all": {}

},

"sort": [ "\_doc" ],

"size": 3

}

{

"\_scroll\_id": "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAACxeFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYBY0b25zVFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAALF8WNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAAACxhFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYhY0b25zVFlWWlRqR3ZJajlfc3BXejJ3",

"took": 5,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 10,

"max\_score": null,

"hits": [

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_score": null,

"\_source": {

"test\_field": "test client 2"

},

"sort": [

0

]

},

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "6",

"\_score": null,

"\_source": {

"test\_field": "tes test"

},

"sort": [

0

]

},

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "AVp4RN0bhjxldOOnBxaE",

"\_score": null,

"\_source": {

"test\_content": "my test"

},

"sort": [

0

]

}

]

}

}

获得的结果会有一个scoll\_id，下一次再发送scoll请求的时候，必须带上这个scoll\_id

GET /\_search/scroll

{

"scroll": "1m",

"scroll\_id" : "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAACxeFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYBY0b25zVFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAALF8WNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAAACxhFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYhY0b25zVFlWWlRqR3ZJajlfc3BXejJ3"

}

11,4,7

3,2,1

20

scoll，看起来挺像分页的，但是其实使用场景不一样。分页主要是用来一页一页搜索，给用户看的；scoll主要是用来一批一批检索数据，让系统进行处理的

## 创建、修改以及删除索引

1、为什么我们要手动创建索引？

2、创建索引

创建索引的语法

PUT /my\_index

{

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

"mappings": {

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

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

...

}

}

创建索引的示例

PUT /my\_index

{

"settings": {

"number\_of\_shards": 1,

"number\_of\_replicas": 0

},

"mappings": {

"my\_type": {

"properties": {

"my\_field": {

"type": "text"

}

}

}

}

}

3、修改索引

PUT /my\_index/\_settings

{

"number\_of\_replicas": 1

}

4、删除索引

DELETE /my\_index

DELETE /index\_one,index\_two

DELETE /index\_\*

DELETE /\_all

这个设置不允许DELETE /\_all 操作

elasticsearch.yml

action.destructive\_requires\_name: true

## 修改分词器以及定制分词器

1、默认的分词器

standard

standard tokenizer：以单词边界进行切分

standard token filter：什么都不做

lowercase token filter：将所有字母转换为小写

stop token filer（默认被禁用）：移除停用词，比如a the it等等

2、修改分词器的设置

启用english停用词token filter

PUT /my\_index

{

"settings": {

"analysis": {

"analyzer": {

"es\_std": {

"type": "standard",

"stopwords": "\_english\_"

}

}

}

}

}

GET /my\_index/\_analyze

{

"analyzer": "standard",

"text": "a dog is in the house"

}

GET /my\_index/\_analyze

{

"analyzer": "es\_std",

"text":"a dog is in the house"

}

3、定制化自己的分词器

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"]

}

}

}

}

}

GET /my\_index/\_analyze

{

"text": "tom&jerry are a friend in the house, <a>, HAHA!!",

"analyzer": "my\_analyzer"

}

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

{

"properties": {

"content": {

"type": "text",

"analyzer": "my\_analyzer"

}

}

}

## type底层数据结构

type，是一个index中用来区分类似的数据的，类似的数据，但是可能有不同的fields，而且有不同的属性来控制索引建立、分词器

field的value，在底层的lucene中建立索引的时候，全部是opaque bytes类型，不区分类型的

lucene是没有type的概念的，在document中，实际上将type作为一个document的field来存储，即\_type，es通过\_type来进行type的过滤和筛选

一个index中的多个type，实际上是放在一起存储的，因此一个index下，不能有多个type重名，而类型或者其他设置不同的，因为那样是无法处理的

{

"ecommerce": {

"mappings": {

"elactronic\_goods": {

"properties": {

"name": {

"type": "string",

},

"price": {

"type": "double"

},

"service\_period": {

"type": "string"

}

}

},

"fresh\_goods": {

"properties": {

"name": {

"type": "string",

},

"price": {

"type": "double"

},

"eat\_period": {

"type": "string"

}

}

}

}

}

}

{

"name": "geli kongtiao",

"price": 1999.0,

"service\_period": "one year"

}

{

"name": "aozhou dalongxia",

"price": 199.0,

"eat\_period": "one week"

}

在底层的存储是这样子的。。。。

{

"ecommerce": {

"mappings": {

"\_type": {

"type": "string",

"index": "not\_analyzed"

},

"name": {

"type": "string"

}

"price": {

"type": "double"

}

"service\_period": {

"type": "string"

}

"eat\_period": {

"type": "string"

}

}

}

}

{

"\_type": "elactronic\_goods",

"name": "geli kongtiao",

"price": 1999.0,

"service\_period": "one year",

"eat\_period": ""

}

{

"\_type": "fresh\_goods",

"name": "aozhou dalongxia",

"price": 199.0,

"service\_period": "",

"eat\_period": "one week"

}

最佳实践，将类似结构的type放在一个index下，这些type应该有多个field是相同的

假如说，你将两个type的field完全不同，放在一个index下，那么就每条数据都至少有一半的field在底层的lucene中是空值，会有严重的性能问题

## mapping root object

1、root object

就是某个type对应的mapping json，包括了properties，metadata（\_id，\_source，\_type），settings（analyzer），其他settings（比如include\_in\_all）

PUT /my\_index

{

"mappings": {

"my\_type": {

"properties": {}

}

}

}

2、properties

type，index，analyzer

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

{

"properties": {

"title": {

"type": "text"

}

}

}

3、\_source

好处:

1. 查询的时候，直接可以拿到完整的document，不需要先拿document id，再发送一次请求拿document
2. partial update基于\_source实现
3. reindex时，直接基于\_source实现，不需要从数据库（或者其他外部存储）查询数据再修改
4. 可以基于\_source定制返回field
5. debug query更容易，因为可以直接看到\_source

如果不需要上述好处，可以禁用\_source

PUT /my\_index/\_mapping/my\_type2

{

"\_source": {"enabled": false}

}

4、\_all

将所有field打包在一起，作为一个\_all field，建立索引。没指定任何field进行搜索时，就是使用\_all field在搜索。

PUT /my\_index/\_mapping/my\_type3

{

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

}

也可以在field级别设置include\_in\_all field，设置是否要将field的值包含在\_all field中

PUT /my\_index/\_mapping/my\_type4

{

"properties": {

"my\_field": {

"type": "text",

"include\_in\_all": false

}

}

}

5、标识性metadata

\_index，\_type，\_id

## 定制化dynamic mapping策略

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 }

}

}

}

## scoll+bulk+索引别名实现零停机重建索引

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": "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAADpAFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAA6QRY0b25zVFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAAOkIWNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAAADpDFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAA6RBY0b25zVFlWWlRqR3ZJajlfc3BXejJ3",

"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" }}

]

}

## 倒排索引组成结构以及其索引可变原因

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写入原理（buffer，segment，commit）

（1）数据写入buffer

（2）commit point

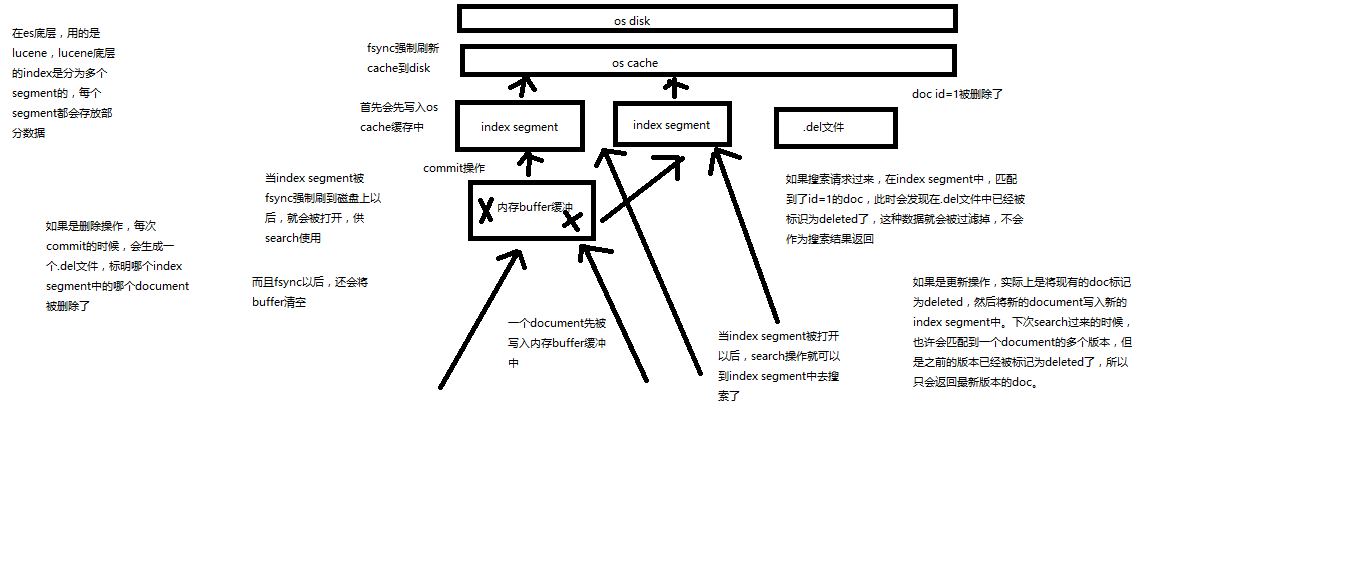
（3）buffer中的数据写入新的index segment

（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

**document增删改内核级原理**

## 优化写入流程实现NRT近实时（filesystem cache，refresh）

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

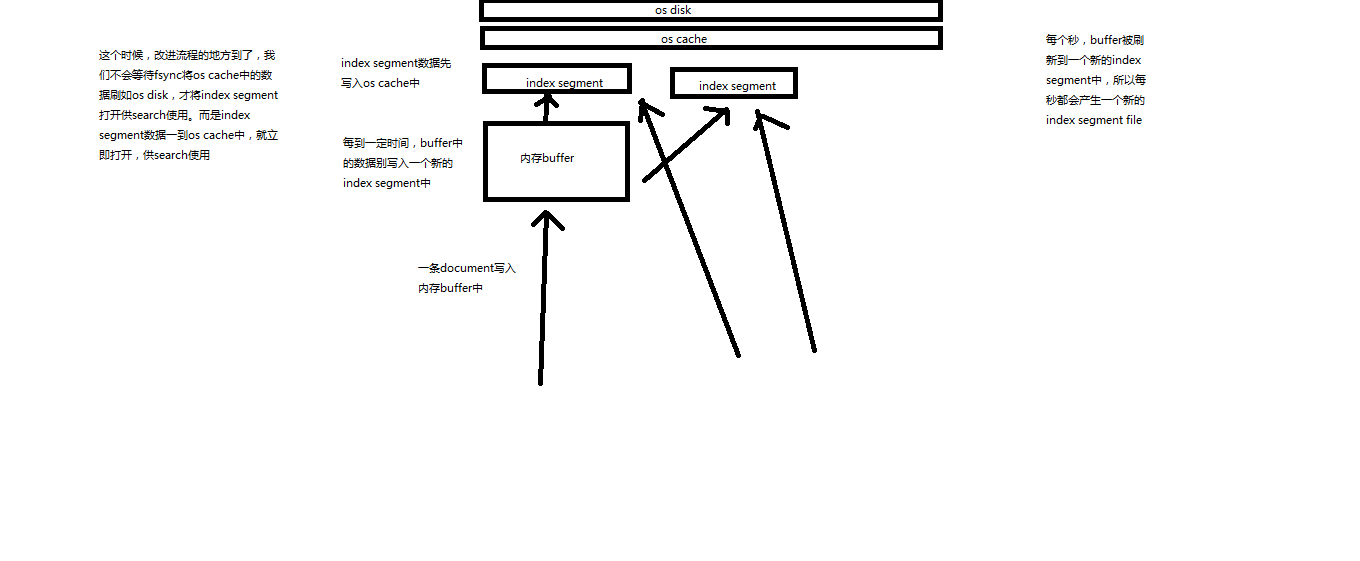
PUT /my\_index

{

"settings": {

"refresh\_interval": "30s"

}

}

**NRT写入优化**

## 继续优化写入流程实现durability可靠存储（translog，flush）

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
  6. buffer中的所有数据写入一个新的segment，并写入os cache，打开供使用
  7. buffer被清空
  8. 一个commit ponit被写入磁盘，标明了所有的index segment
  9. filesystem cache中的所有index segment file缓存数据，被fsync强行刷到磁盘上
  10. 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

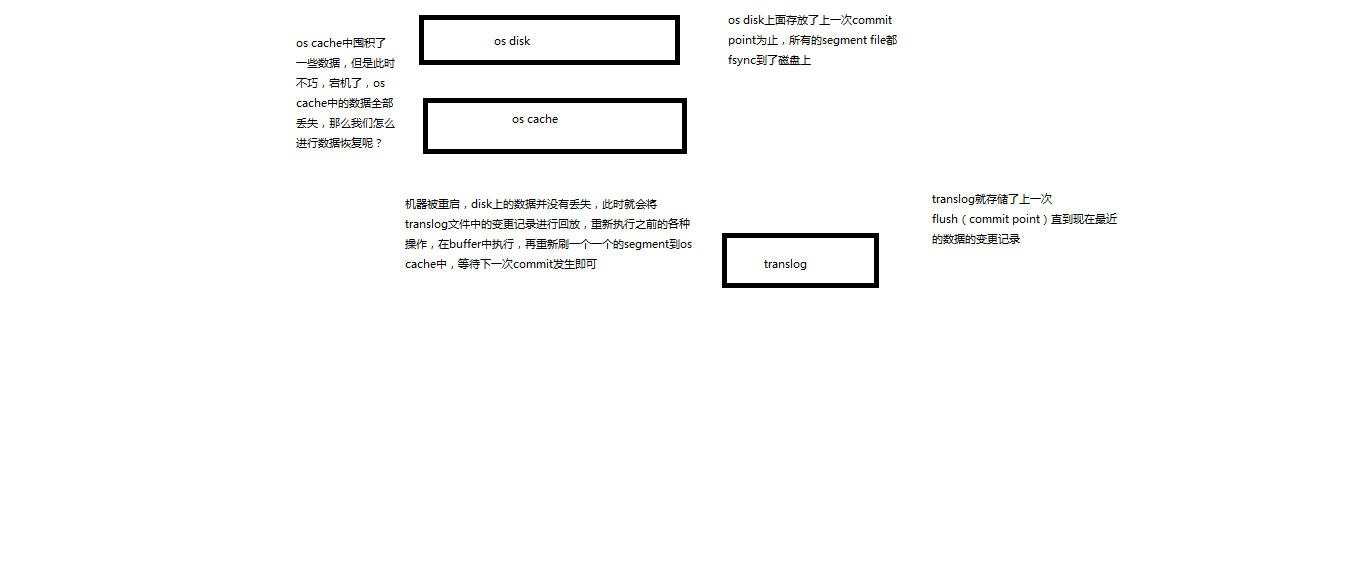
但是这种在一次增删改时强行fsync translog可能会导致部分操作比较耗时，也可以允许部分数据丢失，设置异步fsync translog

PUT /my\_index/\_settings

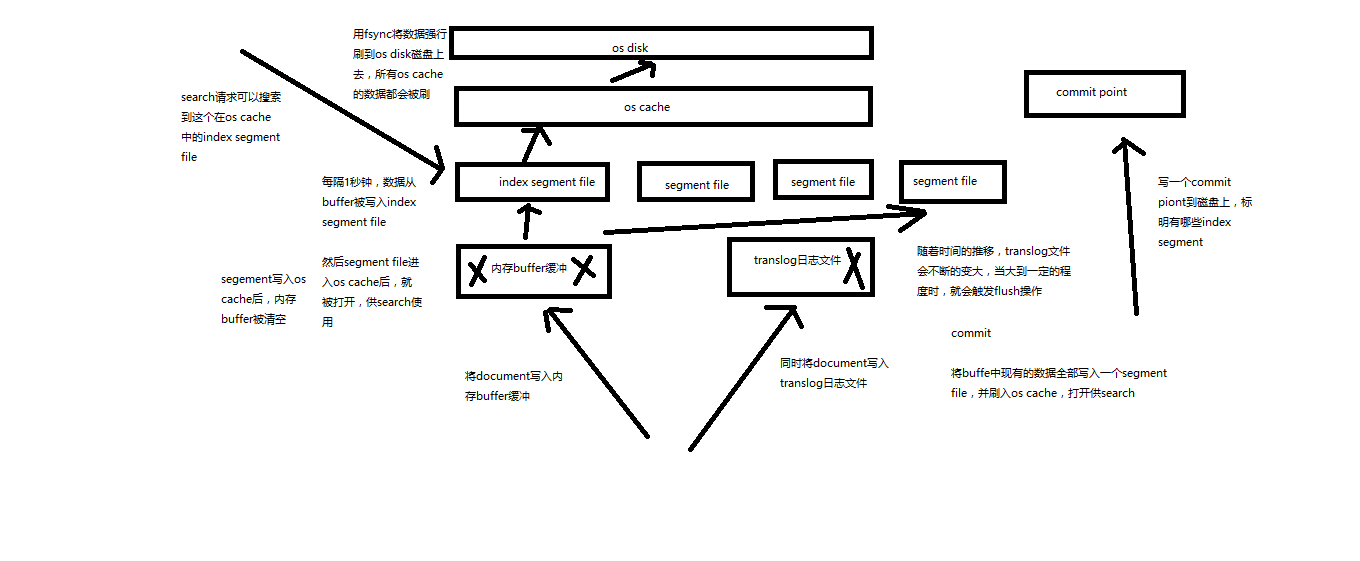
{

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

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

}

**commit point+translog进行数据恢复**

**终极版本的es数据写入流程**

## 优化写入流程实现海量磁盘文件合并（segment merge，optimize）

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

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

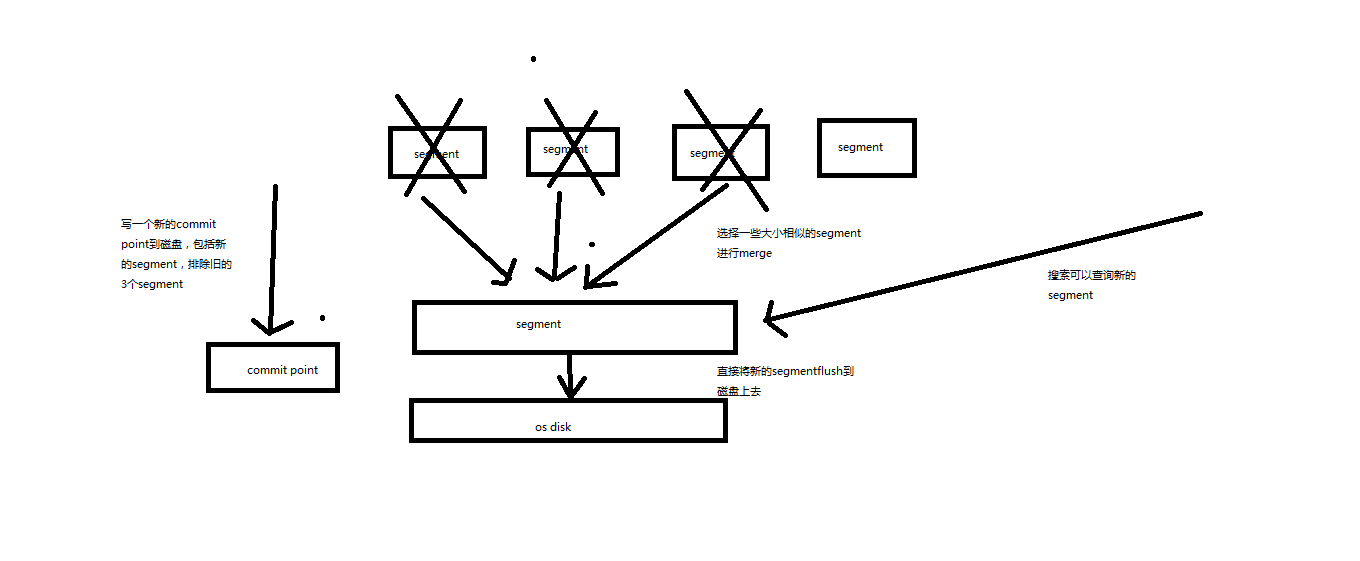
（1）选择一些有相似大小的segment，merge成一个大的segment

（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

POST /my\_index/\_optimize?max\_num\_segments=1，尽量不要手动执行，让它自动默认执行就可以了

**segment merge**

## 基于Java实现员工信息的增删改查

员工信息

姓名

年龄

职位

国家

入职日期

薪水

1、maven依赖

<dependency>

<groupId>org.elasticsearch.client</groupId>

<artifactId>transport</artifactId>

<version>5.2.2</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-api</artifactId>

<version>2.7</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-core</artifactId>

<version>2.7</version>

</dependency>

log4j2.properties

appender.console.type = Console

appender.console.name = console

appender.console.layout.type = PatternLayout

rootLogger.level = info

rootLogger.appenderRef.console.ref = console

2、构建client

Settings settings = Settings.builder()

.put("cluster.name", "myClusterName").build();

TransportClient client = new PreBuiltTransportClient(settings);

TransportClient client = new PreBuiltTransportClient(Settings.EMPTY)

.addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("host1"), 9300))

.addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("host2"), 9300));

client.close();

3、创建document

IndexResponse response = client.prepareIndex("index", "type", "1")

.setSource(jsonBuilder()

.startObject()

.field("user", "kimchy")

.field("postDate", new Date())

.field("message", "trying out Elasticsearch")

.endObject()

)

.get();

4、查询document

GetResponse response = client.prepareGet("index", "type", "1").get();

5、修改document

client.prepareUpdate("index", "type", "1")

.setDoc(jsonBuilder()

.startObject()

.field("gender", "male")

.endObject())

.get();

6、删除document

DeleteResponse response = client.prepareDelete("index", "type", "1").get();

[基于Java实现员工信息的增删改查-代码](file:///I:\项目资料\elasticsearch\基于Java实现员工信息的增删改查-代码)

## 基于Java对员工信息进行复杂的搜索操作

SearchResponse response = client.prepareSearch("index1", "index2")

.setTypes("type1", "type2")

.setQuery(QueryBuilders.termQuery("multi", "test")) // Query

.setPostFilter(QueryBuilders.rangeQuery("age").from(12).to(18)) // Filter

.setFrom(0).setSize(60)

.get();

需求：

（1）搜索职位中包含technique的员工

（2）同时要求age在30到40岁之间

（3）分页查询，查找第一页

GET /company/employee/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"position": "technique"

}

}

],

"filter": {

"range": {

"age": {

"gte": 30,

"lte": 40

}

}

}

}

},

"from": 0,

"size": 1

}

告诉大家，为什么刚才一边运行创建document，一边搜索什么都没搜索到？？？？

近实时！！！

默认是1秒以后，写入es的数据，才能被搜索到。很明显刚才，写入数据不到一秒，我门就在搜索。

<基于Java对员工信息进行复杂的搜索操作-代码>

## 基于Java对员工信息进行聚合分析

SearchResponse sr = node.client().prepareSearch()

.addAggregation(

AggregationBuilders.terms("by\_country").field("country")

.subAggregation(AggregationBuilders.dateHistogram("by\_year")

.field("dateOfBirth")

.dateHistogramInterval(DateHistogramInterval.YEAR)

.subAggregation(AggregationBuilders.avg("avg\_children").field("children"))

)

)

.execute().actionGet();

我们先给个需求：

（1）首先按照country国家来进行分组

（2）然后在每个country分组内，再按照入职年限进行分组

（3）最后计算每个分组内的平均薪资

PUT /company

{

"mappings": {

"employee": {

"properties": {

"age": {

"type": "long"

},

"country": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

},

"fielddata": true

},

"join\_date": {

"type": "date"

},

"name": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"position": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"salary": {

"type": "long"

}

}

}

}

}

GET /company/employee/\_search

{

"size": 0,

"aggs": {

"group\_by\_country": {

"terms": {

"field": "country"

},

"aggs": {

"group\_by\_join\_date": {

"date\_histogram": {

"field": "join\_date",

"interval": "year"

},

"aggs": {

"avg\_salary": {

"avg": {

"field": "salary"

}

}

}

}

}

}

}

}

Map<String, Aggregation> aggrMap = searchResponse.getAggregations().asMap();

StringTerms groupByCountry = (StringTerms) aggrMap.get("group\_by\_country");

Iterator<Bucket> groupByCountryBucketIterator = groupByCountry.getBuckets().iterator();

while(groupByCountryBucketIterator.hasNext()) {

Bucket groupByCountryBucket = groupByCountryBucketIterator.next();

System.out.println(groupByCountryBucket.getKey() + "\t" + groupByCountryBucket.getDocCount());

Histogram groupByJoinDate = (Histogram) groupByCountryBucket.getAggregations().asMap().get("group\_by\_join\_date");

Iterator<org.elasticsearch.search.aggregations.bucket.histogram.Histogram.Bucket> groupByJoinDateBucketIterator = groupByJoinDate.getBuckets().iterator();

while(groupByJoinDateBucketIterator.hasNext()) {

org.elasticsearch.search.aggregations.bucket.histogram.Histogram.Bucket groupByJoinDateBucket = groupByJoinDateBucketIterator.next();

System.out.println(groupByJoinDateBucket.getKey() + "\t" + groupByJoinDateBucket.getDocCount());

Avg avgSalary = (Avg) groupByJoinDateBucket.getAggregations().asMap().get("avg\_salary");

System.out.println(avgSalary.getValue());

}

}

client.close();

}

<基于Java对员工信息进行聚合分析-代码>

## 使用term filter来搜索数据

1. 根据用户ID、是否隐藏、帖子ID、发帖日期来搜索帖子

（1）插入一些测试帖子数据

POST /forum/article/\_bulk

{ "index": { "\_id": 1 }}

{ "articleID" : "XHDK-A-1293-#fJ3", "userID" : 1, "hidden": false, "postDate": "2017-01-01" }

{ "index": { "\_id": 2 }}

{ "articleID" : "KDKE-B-9947-#kL5", "userID" : 1, "hidden": false, "postDate": "2017-01-02" }

{ "index": { "\_id": 3 }}

{ "articleID" : "JODL-X-1937-#pV7", "userID" : 2, "hidden": false, "postDate": "2017-01-01" }

{ "index": { "\_id": 4 }}

{ "articleID" : "QQPX-R-3956-#aD8", "userID" : 2, "hidden": true, "postDate": "2017-01-02" }

初步来说，就先搞4个字段，因为整个es是支持json document格式的，所以说扩展性和灵活性非常之好。如果后续随着业务需求的增加，要在document中增加更多的field，那么我们可以很方便的随时添加field。但是如果是在关系型数据库中，比如mysql，我们建立了一个表，现在要给表中新增一些column，那就很坑爹了，必须用复杂的修改表结构的语法去执行。而且可能对系统代码还有一定的影响。

GET /forum/\_mapping/article

{

"forum": {

"mappings": {

"article": {

"properties": {

"articleID": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"hidden": {

"type": "boolean"

},

"postDate": {

"type": "date"

},

"userID": {

"type": "long"

}

}

}

}

}

}

现在es 5.2版本，type=text，默认会设置两个field，一个是field本身，比如articleID，就是分词的；还有一个的话，就是field.keyword，articleID.keyword，默认不分词，会最多保留256个字符

（2）根据用户ID搜索帖子

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"userID" : 1

}

}

}

}

}

term filter/query：对搜索文本不分词，直接拿去倒排索引中匹配，你输入的是什么，就去匹配什么

比如说，如果对搜索文本进行分词的话，“helle world” --> “hello”和“world”，两个词分别去倒排索引中匹配

term，“hello world” --> “hello world”，直接去倒排索引中匹配“hello world”

（3）搜索没有隐藏的帖子

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"hidden" : false

}

}

}

}

}

（4）根据发帖日期搜索帖子

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"postDate" : "2017-01-01"

}

}

}

}

}

（5）根据帖子ID搜索帖子

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"articleID" : "XHDK-A-1293-#fJ3"

}

}

}

}

}

{

"took": 1,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 0,

"max\_score": null,

"hits": []

}

}

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"articleID.keyword" : "XHDK-A-1293-#fJ3"

}

}

}

}

}

{

"took": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1,

"hits": [

{

"\_index": "forum",

"\_type": "article",

"\_id": "1",

"\_score": 1,

"\_source": {

"articleID": "XHDK-A-1293-#fJ3",

"userID": 1,

"hidden": false,

"postDate": "2017-01-01"

}

}

]

}

}

articleID.keyword，是es最新版本内置建立的field，就是不分词的。所以一个articleID过来的时候，会建立两次索引，一次是自己本身，是要分词的，分词后放入倒排索引；另外一次是基于articleID.keyword，不分词，保留256个字符最多，直接一个字符串放入倒排索引中。

所以term filter，对text过滤，可以考虑使用内置的field.keyword来进行匹配。但是有个问题，默认就保留256个字符。所以尽可能还是自己去手动建立索引，指定not\_analyzed吧。在最新版本的es中，不需要指定not\_analyzed也可以，将type=keyword即可。

（6）查看分词

GET /forum/\_analyze

{

"field": "articleID",

"text": "XHDK-A-1293-#fJ3"

}

默认是analyzed的text类型的field，建立倒排索引的时候，就会对所有的articleID分词，分词以后，原本的articleID就没有了，只有分词后的各个word存在于倒排索引中。

term，是不对搜索文本分词的，XHDK-A-1293-#fJ3 --> XHDK-A-1293-#fJ3；但是articleID建立索引的时候，XHDK-A-1293-#fJ3 --> xhdk，a，1293，fj3

（7）重建索引

DELETE /forum

PUT /forum

{

"mappings": {

"article": {

"properties": {

"articleID": {

"type": "keyword"

}

}

}

}

}

POST /forum/article/\_bulk

{ "index": { "\_id": 1 }}

{ "articleID" : "XHDK-A-1293-#fJ3", "userID" : 1, "hidden": false, "postDate": "2017-01-01" }

{ "index": { "\_id": 2 }}

{ "articleID" : "KDKE-B-9947-#kL5", "userID" : 1, "hidden": false, "postDate": "2017-01-02" }

{ "index": { "\_id": 3 }}

{ "articleID" : "JODL-X-1937-#pV7", "userID" : 2, "hidden": false, "postDate": "2017-01-01" }

{ "index": { "\_id": 4 }}

{ "articleID" : "QQPX-R-3956-#aD8", "userID" : 2, "hidden": true, "postDate": "2017-01-02" }

（8）重新根据帖子ID和发帖日期进行搜索

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"articleID" : "XHDK-A-1293-#fJ3"

}

}

}

}

}

1. 梳理学到的知识点
   1. term filter：根据exact value进行搜索，数字、boolean、date天然支持
   2. text需要建索引时指定为not\_analyzed，才能用term query
   3. 相当于SQL中的单个where条件

select \*

from forum.article

where articleID='XHDK-A-1293-#fJ3'

## filter执行原理深度剖析（bitset机制与caching机制）

1. 在倒排索引中查找搜索串，获取document list

date来举例

word doc1 doc2 doc3

2017-01-01 \* \*

2017-02-02 \* \*

2017-03-03 \* \* \*

filter：2017-02-02

到倒排索引中一找，发现2017-02-02对应的document list是doc2,doc3

1. 为每个在倒排索引中搜索到的结果，构建一个bitset，[0, 0, 0, 1, 0, 1]

非常重要

使用找到的doc list，构建一个bitset，就是一个二进制的数组，数组每个元素都是0或1，用来标识一个doc对一个filter条件是否匹配，如果匹配就是1，不匹配就是0

[0, 1, 1]

doc1：不匹配这个filter的

doc2和do3：是匹配这个filter的

尽可能用简单的数据结构去实现复杂的功能，可以节省内存空间，提升性能

1. 遍历每个过滤条件对应的bitset，优先从最稀疏的开始搜索，查找满足所有条件的document

后面会讲解，一次性其实可以在一个search请求中，发出多个filter条件，每个filter条件都会对应一个bitset

遍历每个filter条件对应的bitset，先从最稀疏的开始遍历

[0, 0, 0, 1, 0, 0]：比较稀疏

[0, 1, 0, 1, 0, 1]

先遍历比较稀疏的bitset，就可以先过滤掉尽可能多的数据

遍历所有的bitset，找到匹配所有filter条件的doc

请求：filter，postDate=2017-01-01，userID=1

postDate: [0, 0, 1, 1, 0, 0]

userID: [0, 1, 0, 1, 0, 1]

遍历完两个bitset之后，找到的匹配所有条件的doc，就是doc4

就可以将document作为结果返回给client了

1. caching bitset，跟踪query，在最近256个query中超过一定次数的过滤条件，缓存其bitset。对于小segment（<1000，或<3%），不缓存bitset。

比如postDate=2017-01-01，[0, 0, 1, 1, 0, 0]，可以缓存在内存中，这样下次如果再有这个条件过来的时候，就不用重新扫描倒排索引，反复生成bitset，可以大幅度提升性能。

在最近的256个filter中，有某个filter超过了一定的次数，次数不固定，就会自动缓存这个filter对应的bitset

segment（上半季），filter针对小segment获取到的结果，可以不缓存，segment记录数<1000，或者segment大小<index总大小的3%

segment数据量很小，此时哪怕是扫描也很快；segment会在后台自动合并，小segment很快就会跟其他小segment合并成大segment，此时就缓存也没有什么意义，segment很快就消失了

针对一个小segment的bitset，[0, 0, 1, 0]

filter比query的好处就在于会caching，但是之前不知道caching的是什么东西，实际上并不是一个filter返回的完整的doc list数据结果。而是filter bitset缓存起来。下次不用扫描倒排索引了。

1. filter大部分情况下来说，在query之前执行，先尽量过滤掉尽可能多的数据

query：是会计算doc对搜索条件的relevance score，还会根据这个score去排序

filter：只是简单过滤出想要的数据，不计算relevance score，也不排序

1. 如果document有新增或修改，那么cached bitset会被自动更新

postDate=2017-01-01，[0, 0, 1, 0]

document，id=5，postDate=2017-01-01，会自动更新到postDate=2017-01-01这个filter的bitset中，全自动，缓存会自动更新。postDate=2017-01-01的bitset，[0, 0, 1, 0, 1]

document，id=1，postDate=2016-12-30，修改为postDate-2017-01-01，此时也会自动更新bitset，[1, 0, 1, 0, 1]

1. 以后只要是有相同的filter条件的，会直接来使用这个过滤条件对应的cached bitset

## 基于bool组合多个filter条件来搜索数据

1. 搜索发帖日期为2017-01-01，或者帖子ID为XHDK-A-1293-#fJ3的帖子，同时要求帖子的发帖日期绝对不为2017-01-02

select \*

from forum.article

where (post\_date='2017-01-01' or article\_id='XHDK-A-1293-#fJ3')

and post\_date!='2017-01-02'

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"bool": {

"should": [

{"term": { "postDate": "2017-01-01" }},

{"term": {"articleID": "XHDK-A-1293-#fJ3"}}

],

"must\_not": {

"term": {

"postDate": "2017-01-02"

}

}

}

}

}

}

}

must，should，must\_not，filter：必须匹配，可以匹配其中任意一个即可，必须不匹配

1. 搜索帖子ID为XHDK-A-1293-#fJ3，或者是帖子ID为JODL-X-1937-#pV7而且发帖日期为2017-01-01的帖子

select \*

from forum.article

where article\_id='XHDK-A-1293-#fJ3'

or (article\_id='JODL-X-1937-#pV7' and post\_date='2017-01-01')

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"bool": {

"should": [

{

"term": {

"articleID": "XHDK-A-1293-#fJ3"

}

},

{

"bool": {

"must": [

{

"term":{

"articleID": "JODL-X-1937-#pV7"

}

},

{

"term": {

"postDate": "2017-01-01"

}

}

]

}

}

]

}

}

}

}

}

1. 梳理学到的知识点

（1）bool：must，must\_not，should，组合多个过滤条件

（2）bool可以嵌套

（3）相当于SQL中的多个and条件：当你把搜索语法学好了以后，基本可以实现部分常用的sql语法对应的功能

## terms搜索多个值以及多值搜索结果优化

term: {"field": "value"}

terms: {"field": ["value1", "value2"]}

sql中的in

select \* from tbl where col in ("value1", "value2")

1. 为帖子数据增加tag字段

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"tag" : ["java", "hadoop"]} }

{ "update": { "\_id": "2"} }

{ "doc" : {"tag" : ["java"]} }

{ "update": { "\_id": "3"} }

{ "doc" : {"tag" : ["hadoop"]} }

{ "update": { "\_id": "4"} }

{ "doc" : {"tag" : ["java", "elasticsearch"]} }

1. 搜索articleID为KDKE-B-9947-#kL5或QQPX-R-3956-#aD8的帖子，搜索tag中包含java的帖子

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"terms": {

"articleID": [

"KDKE-B-9947-#kL5",

"QQPX-R-3956-#aD8"

]

}

}

}

}

}

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"terms" : {

"tag" : ["java"]

}

}

}

}

}

"took": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 1,

"hits": [

{

"\_index": "forum",

"\_type": "article",

"\_id": "2",

"\_score": 1,

"\_source": {

"articleID": "KDKE-B-9947-#kL5",

"userID": 1,

"hidden": false,

"postDate": "2017-01-02",

"tag": [

"java"

]

}

},

{

"\_index": "forum",

"\_type": "article",

"\_id": "4",

"\_score": 1,

"\_source": {

"articleID": "QQPX-R-3956-#aD8",

"userID": 2,

"hidden": true,

"postDate": "2017-01-02",

"tag": [

"java",

"elasticsearch"

]

}

},

{

"\_index": "forum",

"\_type": "article",

"\_id": "1",

"\_score": 1,

"\_source": {

"articleID": "XHDK-A-1293-#fJ3",

"userID": 1,

"hidden": false,

"postDate": "2017-01-01",

"tag": [

"java",

"hadoop"

]

}

}

]

}

}

1. 优化搜索结果，仅仅搜索tag只包含java的帖子

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"tag\_cnt" : 2} }

{ "update": { "\_id": "2"} }

{ "doc" : {"tag\_cnt" : 1} }

{ "update": { "\_id": "3"} }

{ "doc" : {"tag\_cnt" : 1} }

{ "update": { "\_id": "4"} }

{ "doc" : {"tag\_cnt" : 2} }

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"bool": {

"must": [

{

"term": {

"tag\_cnt": 1

}

},

{

"terms": {

"tag": ["java"]

}

}

]

}

}

}

}

}

["java", "hadoop", "elasticsearch"]

1. 学到的知识点梳理

（1）terms多值搜索

（2）优化terms多值搜索的结果

（3）相当于SQL中的in语句

## range filter来进行范围过滤

1. 为帖子数据增加浏览量的字段

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"view\_cnt" : 30} }

{ "update": { "\_id": "2"} }

{ "doc" : {"view\_cnt" : 50} }

{ "update": { "\_id": "3"} }

{ "doc" : {"view\_cnt" : 100} }

{ "update": { "\_id": "4"} }

{ "doc" : {"view\_cnt" : 80} }

1. 搜索浏览量在30~60之间的帖子

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"view\_cnt": {

"gt": 30,

"lt": 60

}

}

}

}

}

}

gte

lte

1. 搜索发帖日期在最近1个月的帖子

POST /forum/article/\_bulk

{ "index": { "\_id": 5 }}

{ "articleID" : "DHJK-B-1395-#Ky5", "userID" : 3, "hidden": false, "postDate": "2017-03-01", "tag": ["elasticsearch"], "tag\_cnt": 1, "view\_cnt": 10 }

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"postDate": {

"gt": "2017-03-10||-30d"

}

}

}

}

}

}

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"postDate": {

"gt": "now-30d"

}

}

}

}

}

}

1. 梳理一下学到的知识点

（1）range，sql中的between，或者是>=1，<=1

（2）range做范围过滤

## 手动控制全文检索结果的精准度

1. 为帖子数据增加标题字段

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"title" : "this is java and elasticsearch blog"} }

{ "update": { "\_id": "2"} }

{ "doc" : {"title" : "this is java blog"} }

{ "update": { "\_id": "3"} }

{ "doc" : {"title" : "this is elasticsearch blog"} }

{ "update": { "\_id": "4"} }

{ "doc" : {"title" : "this is java, elasticsearch, hadoop blog"} }

{ "update": { "\_id": "5"} }

{ "doc" : {"title" : "this is spark blog"} }

1. 搜索标题中包含java或elasticsearch的blog

这个，就跟之前的那个term query，不一样了。不是搜索exact value，是进行full text全文检索。

match query，是负责进行全文检索的。当然，如果要检索的field，是not\_analyzed类型的，那么match query也相当于term query。

GET /forum/article/\_search

{

"query": {

"match": {

"title": "java elasticsearch"

}

}

}

1. 搜索标题中包含java和elasticsearch的blog

搜索结果精准控制的第一步：灵活使用and关键字，如果你是希望所有的搜索关键字都要匹配的，那么就用and，可以实现单纯match query无法实现的效果

GET /forum/article/\_search

{

"query": {

"match": {

"title": {

"query": "java elasticsearch",

"operator": "and"

}

}

}

}

1. 搜索包含java，elasticsearch，spark，hadoop，4个关键字中，至少3个的blog

控制搜索结果的精准度的第二步：指定一些关键字中，必须至少匹配其中的多少个关键字，才能作为结果返回

GET /forum/article/\_search

{

"query": {

"match": {

"title": {

"query": "java elasticsearch spark hadoop",

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

}

}

}

}

1. 用bool组合多个搜索条件，来搜索title

GET /forum/article/\_search

{

"query": {

"bool": {

"must": { "match": { "title": "java" }},

"must\_not": { "match": { "title": "spark" }},

"should": [

{ "match": { "title": "hadoop" }},

{ "match": { "title": "elasticsearch" }}

]

}

}

}

1. bool组合多个搜索条件，如何计算relevance score

must和should搜索对应的分数，加起来，除以must和should的总数

排名第一：java，同时包含should中所有的关键字，hadoop，elasticsearch

排名第二：java，同时包含should中的elasticsearch

排名第三：java，不包含should中的任何关键字

should是可以影响相关度分数的

must是确保说，谁必须有这个关键字，同时会根据这个must的条件去计算出document对这个搜索条件的relevance score

在满足must的基础之上，should中的条件，不匹配也可以，但是如果匹配的更多，那么document的relevance score就会更高

{

"took": 6,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 1.3375794,

"hits": [

{

"\_index": "forum",

"\_type": "article",

"\_id": "4",

"\_score": 1.3375794,

"\_source": {

"articleID": "QQPX-R-3956-#aD8",

"userID": 2,

"hidden": true,

"postDate": "2017-01-02",

"tag": [

"java",

"elasticsearch"

],

"tag\_cnt": 2,

"view\_cnt": 80,

"title": "this is java, elasticsearch, hadoop blog"

}

},

{

"\_index": "forum",

"\_type": "article",

"\_id": "1",

"\_score": 0.53484553,

"\_source": {

"articleID": "XHDK-A-1293-#fJ3",

"userID": 1,

"hidden": false,

"postDate": "2017-01-01",

"tag": [

"java",

"hadoop"

],

"tag\_cnt": 2,

"view\_cnt": 30,

"title": "this is java and elasticsearch blog"

}

},

{

"\_index": "forum",

"\_type": "article",

"\_id": "2",

"\_score": 0.19856805,

"\_source": {

"articleID": "KDKE-B-9947-#kL5",

"userID": 1,

"hidden": false,

"postDate": "2017-01-02",

"tag": [

"java"

],

"tag\_cnt": 1,

"view\_cnt": 50,

"title": "this is java blog"

}

}

]

}

}

1. 搜索java，hadoop，spark，elasticsearch，至少包含其中3个关键字

默认情况下，should是可以不匹配任何一个的，比如上面的搜索中，this is java blog，就不匹配任何一个should条件

但是有个例外的情况，如果没有must的话，那么should中必须至少匹配一个才可以

比如下面的搜索，should中有4个条件，默认情况下，只要满足其中一个条件，就可以匹配作为结果返回

但是可以精准控制，should的4个条件中，至少匹配几个才能作为结果返回

GET /forum/article/\_search

{

"query": {

"bool": {

"should": [

{ "match": { "title": "java" }},

{ "match": { "title": "elasticsearch" }},

{ "match": { "title": "hadoop" }},

{ "match": { "title": "spark" }}

],

"minimum\_should\_match": 3

}

}

}

1. 梳理一下学到的知识点

1、全文检索的时候，进行多个值的检索，有两种做法，match query；should

2、控制搜索结果精准度：and operator，minimum\_should\_match

## term+bool实现的multiword搜索底层原理剖析

1. 普通match如何转换为term+should

{

"match": { "title": "java elasticsearch"}

}

使用诸如上面的match query进行多值搜索的时候，es会在底层自动将这个match query转换为bool的语法

bool should，指定多个搜索词，同时使用term query

{

"bool": {

"should": [

{ "term": { "title": "java" }},

{ "term": { "title": "elasticsearch" }}

]

}

}

1. and match如何转换为term+must

{

"match": {

"title": {

"query": "java elasticsearch",

"operator": "and"

}

}

}

{

"bool": {

"must": [

{ "term": { "title": "java" }},

{ "term": { "title": "elasticsearch" }}

]

}

}

1. minimum\_should\_match如何转换

{

"match": {

"title": {

"query": "java elasticsearch hadoop spark",

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

}

}

}

{

"bool": {

"should": [

{ "term": { "title": "java" }},

{ "term": { "title": "elasticsearch" }},

{ "term": { "title": "hadoop" }},

{ "term": { "title": "spark" }}

],

"minimum\_should\_match": 3

}

}

## boost的细粒度搜索条件权重控制

需求：搜索标题中包含java的帖子，同时呢，如果标题中包含hadoop或elasticsearch就优先搜索出来，同时呢，如果一个帖子包含java hadoop，一个帖子包含java elasticsearch，包含hadoop的帖子要比elasticsearch优先搜索出来

知识点，搜索条件的权重，boost，可以将某个搜索条件的权重加大，此时当匹配这个搜索条件和匹配另一个搜索条件的document，计算relevance score时，匹配权重更大的搜索条件的document，relevance score会更高，当然也就会优先被返回回来

默认情况下，搜索条件的权重都是一样的，都是1

GET /forum/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"title": "blog"

}

}

],

"should": [

{

"match": {

"title": {

"query": "java"

}

}

},

{

"match": {

"title": {

"query": "hadoop"

}

}

},

{

"match": {

"title": {

"query": "elasticsearch"

}

}

},

{

"match": {

"title": {

"query": "spark",

"boost": 5

}

}

}

]

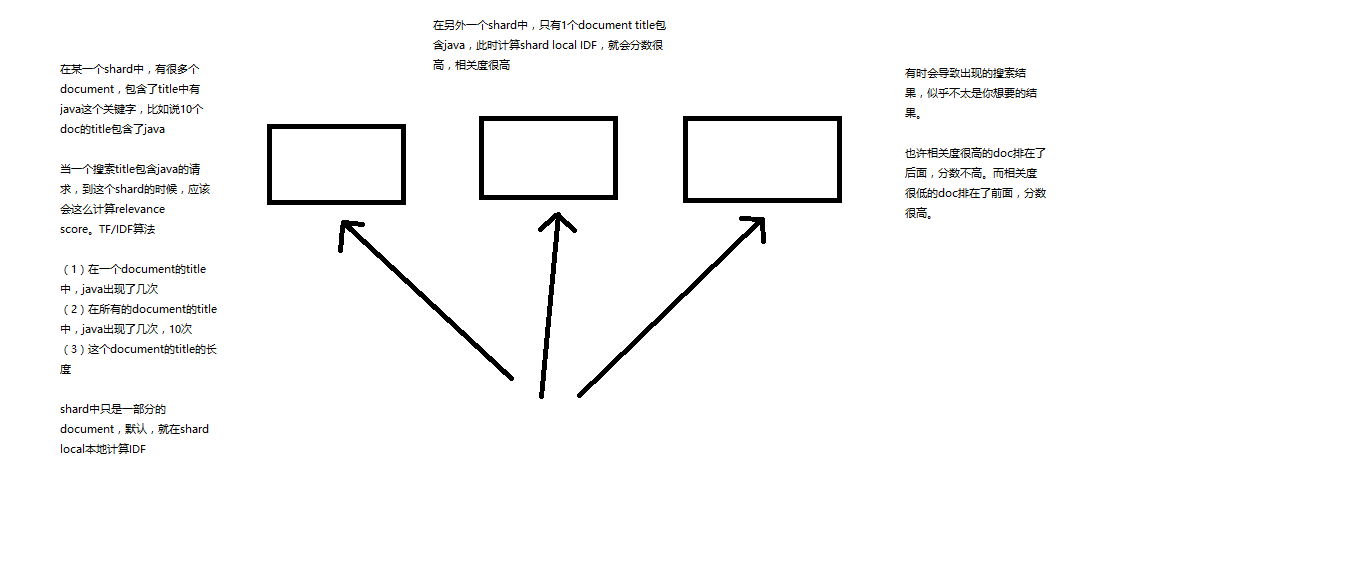
}

}

}

## 多shard场景下relevance score不准确问题

* + 1. 多shard场景下relevance score不准确问题

图解

**多shard场景下搜素结果不准确**

* + 1. 如何解决该问题？

1. 生产环境下，数据量大，尽可能实现均匀分配

数据量很大的话，其实一般情况下，在概率学的背景下，es都是在多个shard中均匀路由数据的，路由的时候根据\_id，负载均衡

比如说有10个document，title都包含java，一共有5个shard，那么在概率学的背景下，如果负载均衡的话，其实每个shard都应该有2个doc，title包含java

如果说数据分布均匀的话，其实就没有刚才说的那个问题了

1. 测试环境下，将索引的primary shard设置为1个，number\_of\_shards=1，index settings

如果说只有一个shard，那么当然，所有的document都在这个shard里面，就没有这个问题了

1. 测试环境下，搜索附带search\_type=dfs\_query\_then\_fetch参数，会将local IDF取出来计算global IDF

计算一个doc的相关度分数的时候，就会将所有shard对的local IDF计算一下，获取出来，在本地进行global IDF分数的计算，会将所有shard的doc作为上下文来进行计算，也能确保准确性。但是production生产环境下，不推荐这个参数，因为性能很差。

## dis\_max实现best fields策略进行多字段搜索

1. 为帖子数据增加content字段

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"content" : "i like to write best elasticsearch article"} }

{ "update": { "\_id": "2"} }

{ "doc" : {"content" : "i think java is the best programming language"} }

{ "update": { "\_id": "3"} }

{ "doc" : {"content" : "i am only an elasticsearch beginner"} }

{ "update": { "\_id": "4"} }

{ "doc" : {"content" : "elasticsearch and hadoop are all very good solution, i am a beginner"} }

{ "update": { "\_id": "5"} }

{ "doc" : {"content" : "spark is best big data solution based on scala ,an programming language similar to java"} }

1. 搜索title或content中包含java或solution的帖子

下面这个就是multi-field搜索，多字段搜索

GET /forum/article/\_search

{

"query": {

"bool": {

"should": [

{ "match": { "title": "java solution" }},

{ "match": { "content": "java solution" }}

]

}

}

}

1. 结果分析

期望的是doc5，结果是doc2,doc4排在了前面

计算每个document的relevance score：每个query的分数，乘以matched query数量，除以总query数量

算一下doc4的分数

{ "match": { "title": "java solution" }}，针对doc4，是有一个分数的

{ "match": { "content": "java solution" }}，针对doc4，也是有一个分数的

所以是两个分数加起来，比如说，1.1 + 1.2 = 2.3

matched query数量 = 2

总query数量 = 2

2.3 \* 2 / 2 = 2.3

算一下doc5的分数

{ "match": { "title": "java solution" }}，针对doc5，是没有分数的

{ "match": { "content": "java solution" }}，针对doc5，是有一个分数的

所以说，只有一个query是有分数的，比如2.3

matched query数量 = 1

总query数量 = 2

2.3 \* 1 / 2 = 1.15

doc5的分数 = 1.15 < doc4的分数 = 2.3

1. best fields策略，dis\_max

best fields策略，就是说，搜索到的结果，应该是某一个field中匹配到了尽可能多的关键词，被排在前面；而不是尽可能多的field匹配到了少数的关键词，排在了前面

dis\_max语法，直接取多个query中，分数最高的那一个query的分数即可

{ "match": { "title": "java solution" }}，针对doc4，是有一个分数的，1.1

{ "match": { "content": "java solution" }}，针对doc4，也是有一个分数的，1.2

取最大分数，1.2

{ "match": { "title": "java solution" }}，针对doc5，是没有分数的

{ "match": { "content": "java solution" }}，针对doc5，是有一个分数的，2.3

取最大分数，2.3

然后doc4的分数 = 1.2 < doc5的分数 = 2.3，所以doc5就可以排在更前面的地方，符合我们的需要

GET /forum/article/\_search

{

"query": {

"dis\_max": {

"queries": [

{ "match": { "title": "java solution" }},

{ "match": { "content": "java solution" }}

]

}

}

}

## tie\_breaker参数优化dis\_max搜索效果

1. 搜索title或content中包含java beginner的帖子

GET /forum/article/\_search

{

"query": {

"dis\_max": {

"queries": [

{ "match": { "title": "java beginner" }},

{ "match": { "body": "java beginner" }}

]

}

}

}

有些场景不是太好复现的，因为是这样，你需要尝试去构造不同的文本，然后去构造一些搜索出来，去达到你要的一个效果

可能在实际场景中出现的一个情况是这样的：

1. 某个帖子，doc1，title中包含java，content不包含java beginner任何一个关键词
2. 某个帖子，doc2，content中包含beginner，title中不包含任何一个关键词
3. 某个帖子，doc3，title中包含java，content中包含beginner
4. 最终搜索，可能出来的结果是，doc1和doc2排在doc3的前面，而不是我们期望的doc3排在最前面dis\_max，只是取分数最高的那个query的分数而已。
5. dis\_max只取某一个query最大的分数，完全不考虑其他query的分数
6. 使用tie\_breaker将其他query的分数也考虑进去

tie\_breaker参数的意义，在于说，将其他query的分数，乘以tie\_breaker，然后综合与最高分数的那个query的分数，综合在一起进行计算

除了取最高分以外，还会考虑其他的query的分数

tie\_breaker的值，在0~1之间，是个小数，就ok

GET /forum/article/\_search

{

"query": {

"dis\_max": {

"queries": [

{ "match": { "title": "java beginner" }},

{ "match": { "body": "java beginner" }}

],

"tie\_breaker": 0.3

}

}

}

## multi\_match语法实现dis\_max+tie\_breaker

"fields":[ "title^2", "content" ]相当于boost ：2 ，把他的权重设置为2.

GET /forum/article/\_search

{

"query": {

"multi\_match": {

"query": "java solution",

"type": "best\_fields",

"fields": [ "title^2", "content" ],

"tie\_breaker": 0.3,

"minimum\_should\_match": "50%"

}

}

}

GET /forum/article/\_search

{

"query": {

"dis\_max": {

"queries": [

{

"match": {

"title": {

"query": "java beginner",

"minimum\_should\_match": "50%",

"boost": 2

}

}

},

{

"match": {

"body": {

"query": "java beginner",

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

}

}

}

],

"tie\_breaker": 0.3

}

}

}

minimum\_should\_match，主要是用来干嘛的？

去长尾，long tail

长尾，比如你搜索5个关键词，但是很多结果是只匹配1个关键词的，其实跟你想要的结果相差甚远，这些结果就是长尾

minimum\_should\_match，控制搜索结果的精准度，只有匹配一定数量的关键词的数据，才能返回

## multi\_match+most fiels策略进行multi-field搜索

从best-fields换成most-fields策略

best-fields策略，主要是说将某一个field匹配尽可能多的关键词的doc优先返回回来

most-fields策略，主要是说尽可能返回更多field匹配到某个关键词的doc，优先返回回来

POST /forum/\_mapping/article

{

"properties": {

"sub\_title": {

"type": "string",

"analyzer": "english",

"fields": {

"std": {

"type": "string",

"analyzer": "standard"

}

}

}

}

}

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"sub\_title" : "learning more courses"} }

{ "update": { "\_id": "2"} }

{ "doc" : {"sub\_title" : "learned a lot of course"} }

{ "update": { "\_id": "3"} }

{ "doc" : {"sub\_title" : "we have a lot of fun"} }

{ "update": { "\_id": "4"} }

{ "doc" : {"sub\_title" : "both of them are good"} }

{ "update": { "\_id": "5"} }

{ "doc" : {"sub\_title" : "haha, hello world"} }

GET /forum/article/\_search

{

"query": {

"match": {

"sub\_title": "learning courses"

}

}

}

{

"took": 3,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 2,

"max\_score": 1.219939,

"hits": [

{

"\_index": "forum",

"\_type": "article",

"\_id": "2",

"\_score": 1.219939,

"\_source": {

"articleID": "KDKE-B-9947-#kL5",

"userID": 1,

"hidden": false,

"postDate": "2017-01-02",

"tag": [

"java"

],

"tag\_cnt": 1,

"view\_cnt": 50,

"title": "this is java blog",

"content": "i think java is the best programming language",

"sub\_title": "learned a lot of course"

}

},

{

"\_index": "forum",

"\_type": "article",

"\_id": "1",

"\_score": 0.5063205,

"\_source": {

"articleID": "XHDK-A-1293-#fJ3",

"userID": 1,

"hidden": false,

"postDate": "2017-01-01",

"tag": [

"java",

"hadoop"

],

"tag\_cnt": 2,

"view\_cnt": 30,

"title": "this is java and elasticsearch blog",

"content": "i like to write best elasticsearch article",

"sub\_title": "learning more courses"

}

}

]

}

}

sub\_title用的是enligsh analyzer，所以还原了单词

为什么，因为如果我们用的是类似于english analyzer这种分词器的话，就会将单词还原为其最基本的形态，stemmer

learning --> learn

learned --> learn

courses --> course

sub\_titile: learning coureses --> learn course

{ "doc" : {"sub\_title" : "learned a lot of course"} }，就排在了{ "doc" : {"sub\_title" : "learning more courses"} }的前面

GET /forum/article/\_search

{

"query": {

"match": {

"sub\_title": "learning courses"

}

}

}

GET /forum/article/\_search

{

"query": {

"multi\_match": {

"query": "learning courses",

"type": "most\_fields",

"fields": [ "sub\_title", "sub\_title.std" ]

}

}

}

{

"took": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 2,

"max\_score": 1.219939,

"hits": [

{

"\_index": "forum",

"\_type": "article",

"\_id": "2",

"\_score": 1.219939,

"\_source": {

"articleID": "KDKE-B-9947-#kL5",

"userID": 1,

"hidden": false,

"postDate": "2017-01-02",

"tag": [

"java"

],

"tag\_cnt": 1,

"view\_cnt": 50,

"title": "this is java blog",

"content": "i think java is the best programming language",

"sub\_title": "learned a lot of course"

}

},

{

"\_index": "forum",

"\_type": "article",

"\_id": "1",

"\_score": 1.012641,

"\_source": {

"articleID": "XHDK-A-1293-#fJ3",

"userID": 1,

"hidden": false,

"postDate": "2017-01-01",

"tag": [

"java",

"hadoop"

],

"tag\_cnt": 2,

"view\_cnt": 30,

"title": "this is java and elasticsearch blog",

"content": "i like to write best elasticsearch article",

"sub\_title": "learning more courses"

}

}

]

}

}

与best\_fields的区别

* 1. best\_fields，是对多个field进行搜索，挑选某个field匹配度最高的那个分数，同时在多个query最高分相同的情况下，在一定程度上考虑其他query的分数。简单来说，你对多个field进行搜索，就想搜索到某一个field尽可能包含更多关键字的数据

优点：通过best\_fields策略，以及综合考虑其他field，还有minimum\_should\_match支持，可以尽可能精准地将匹配的结果推送到最前面

缺点：除了那些精准匹配的结果，其他差不多大的结果，排序结果不是太均匀，没有什么区分度了

实际的例子：百度之类的搜索引擎，最匹配的到最前面，但是其他的就没什么区分度了

* 1. most\_fields，综合多个field一起进行搜索，尽可能多地让所有field的query参与到总分数的计算中来，此时就会是个大杂烩，出现类似best\_fields案例最开始的那个结果，结果不一定精准，某一个document的一个field包含更多的关键字，但是因为其他document有更多field匹配到了，所以排在了前面；所以需要建立类似sub\_title.std这样的field，尽可能让某一个field精准匹配query string，贡献更高的分数，将更精准匹配的数据排到前面

优点：将尽可能匹配更多field的结果推送到最前面，整个排序结果是比较均匀的

缺点：可能那些精准匹配的结果，无法推送到最前面

实际的例子：wiki，明显的most\_fields策略，搜索结果比较均匀，但是的确要翻好几页才能找到最匹配的结果

## most\_fields策略进行cross-fields search弊端

cross-fields搜索，一个唯一标识，跨了多个field。比如一个人，标识，是姓名；一个建筑，它的标识是地址。姓名可以散落在多个field中，比如first\_name和last\_name中，地址可以散落在country，province，city中。

跨多个field搜索一个标识，比如搜索一个人名，或者一个地址，就是cross-fields搜索

初步来说，如果要实现，可能用most\_fields比较合适。因为best\_fields是优先搜索单个field最匹配的结果，cross-fields本身就不是一个field的问题了。

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"author\_first\_name" : "Peter", "author\_last\_name" : "Smith"} }

{ "update": { "\_id": "2"} }

{ "doc" : {"author\_first\_name" : "Smith", "author\_last\_name" : "Williams"} }

{ "update": { "\_id": "3"} }

{ "doc" : {"author\_first\_name" : "Jack", "author\_last\_name" : "Ma"} }

{ "update": { "\_id": "4"} }

{ "doc" : {"author\_first\_name" : "Robbin", "author\_last\_name" : "Li"} }

{ "update": { "\_id": "5"} }

{ "doc" : {"author\_first\_name" : "Tonny", "author\_last\_name" : "Peter Smith"} }

GET /forum/article/\_search

{

"query": {

"multi\_match": {

"query": "Peter Smith",

"type": "most\_fields",

"fields": [ "author\_first\_name", "author\_last\_name" ]

}

}

}

Peter Smith，匹配author\_first\_name，匹配到了Smith，这时候它的分数很高，为什么啊？？？

因为IDF分数高，IDF分数要高，那么这个匹配到的term（Smith），在所有doc中的出现频率要低，author\_first\_name field中，Smith就出现过1次

Peter Smith这个人，doc 1，Smith在author\_last\_name中，但是author\_last\_name出现了两次Smith，所以导致doc 1的IDF分数较低

{

"took": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 0.6931472,

"hits": [

{

"\_index": "forum",

"\_type": "article",

"\_id": "2",

"\_score": 0.6931472,

"\_source": {

"articleID": "KDKE-B-9947-#kL5",

"userID": 1,

"hidden": false,

"postDate": "2017-01-02",

"tag": [

"java"

],

"tag\_cnt": 1,

"view\_cnt": 50,

"title": "this is java blog",

"content": "i think java is the best programming language",

"sub\_title": "learned a lot of course",

"author\_first\_name": "Smith",

"author\_last\_name": "Williams"

}

},

{

"\_index": "forum",

"\_type": "article",

"\_id": "1",

"\_score": 0.5753642,

"\_source": {

"articleID": "XHDK-A-1293-#fJ3",

"userID": 1,

"hidden": false,

"postDate": "2017-01-01",

"tag": [

"java",

"hadoop"

],

"tag\_cnt": 2,

"view\_cnt": 30,

"title": "this is java and elasticsearch blog",

"content": "i like to write best elasticsearch article",

"sub\_title": "learning more courses",

"author\_first\_name": "Peter",

"author\_last\_name": "Smith"

}

},

{

"\_index": "forum",

"\_type": "article",

"\_id": "5",

"\_score": 0.51623213,

"\_source": {

"articleID": "DHJK-B-1395-#Ky5",

"userID": 3,

"hidden": false,

"postDate": "2017-03-01",

"tag": [

"elasticsearch"

],

"tag\_cnt": 1,

"view\_cnt": 10,

"title": "this is spark blog",

"content": "spark is best big data solution based on scala ,an programming language similar to java",

"sub\_title": "haha, hello world",

"author\_first\_name": "Tonny",

"author\_last\_name": "Peter Smith"

}

}

]

}

}

问题1：只是找到尽可能多的field匹配的doc，而不是某个field完全匹配的doc

问题2：most\_fields，没办法用minimum\_should\_match去掉长尾数据，就是匹配的特别少的结果

问题3：TF/IDF算法，比如Peter Smith和Smith Williams，搜索Peter Smith的时候，由于first\_name中很少有Smith的，所以query在所有document中的频率很低，得到的分数很高，可能Smith Williams反而会排在Peter Smith前面

## copy\_to定制组合field解决cross-fields搜索弊端

copy\_to:将字段组合起来。指向的字段不会在head插件查看时显示，但是能通过查询语句作为条件.

most\_fields策略，去实现cross-fields搜索，有3大弊端，而且搜索结果也显示出了这3大弊端

第一个办法：用copy\_to，将多个field组合成一个field

问题其实就出在有多个field，有多个field以后，就很尴尬，我们只要想办法将一个标识跨在多个field的情况，合并成一个field即可。比如说，一个人名，本来是first\_name，last\_name，现在合并成一个full\_name，不就ok了吗。。。。。

PUT /forum/\_mapping/article

{

"properties": {

"new\_author\_first\_name": {

"type": "string",

"copy\_to": "new\_author\_full\_name"

},

"new\_author\_last\_name": {

"type": "string",

"copy\_to": "new\_author\_full\_name"

},

"new\_author\_full\_name": {

"type": "string"

}

}

}

用了这个copy\_to语法之后，就可以将多个字段的值拷贝到一个字段中，并建立倒排索引

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"new\_author\_first\_name" : "Peter", "new\_author\_last\_name" : "Smith"} } --> Peter Smith

{ "update": { "\_id": "2"} }

{ "doc" : {"new\_author\_first\_name" : "Smith", "new\_author\_last\_name" : "Williams"} } --> Smith Williams

{ "update": { "\_id": "3"} }

{ "doc" : {"new\_author\_first\_name" : "Jack", "new\_author\_last\_name" : "Ma"} } --> Jack Ma

{ "update": { "\_id": "4"} }

{ "doc" : {"new\_author\_first\_name" : "Robbin", "new\_author\_last\_name" : "Li"} } --> Robbin Li

{ "update": { "\_id": "5"} }

{ "doc" : {"new\_author\_first\_name" : "Tonny", "new\_author\_last\_name" : "Peter Smith"} } --> Tonny Peter Smith

GET /forum/article/\_search

{

"query": {

"match": {

"new\_author\_full\_name": "Peter Smith"

}

}

}

很无奈，很多时候，我们很难复现。比如官网也会给一些例子，说用什么什么文本，怎么怎么搜索，是怎么怎么样的效果。es版本在不断迭代，这个打分的算法也在不断的迭代。所以我们其实很难说，对类似这几讲讲解的best\_fields，most\_fields，cross\_fields，完全复现出来应有的场景和效果。

更多的把原理和知识点给大家讲解清楚，带着大家演练一遍怎么操作的，做一下实验

期望的是说，比如大家自己在开发搜索应用的时候，碰到需要best\_fields的场景，知道怎么做，知道best\_fields的原理，可以达到什么效果；碰到most\_fields的场景，知道怎么做，以及原理；碰到搜搜cross\_fields标识的场景，知道怎么做，知道原理是什么，效果是什么。。。。

问题1：只是找到尽可能多的field匹配的doc，而不是某个field完全匹配的doc --> 解决，最匹配的document被最先返回

问题2：most\_fields，没办法用minimum\_should\_match去掉长尾数据，就是匹配的特别少的结果 --> 解决，可以使用minimum\_should\_match去掉长尾数据

问题3：TF/IDF算法，比如Peter Smith和Smith Williams，搜索Peter Smith的时候，由于first\_name中很少有Smith的，所以query在所有document中的频率很低，得到的分数很高，可能Smith Williams反而会排在Peter Smith前面 --> 解决，Smith和Peter在一个field了，所以在所有document中出现的次数是均匀的，不会有极端的偏差

## 原生cross-fiels技术解决搜索弊端

GET /forum/article/\_search

{

"query": {

"multi\_match": {

"query": "Peter Smith",

"type": "cross\_fields",

"operator": "and",

"fields": ["author\_first\_name", "author\_last\_name"]

}

}

}

问题1：只是找到尽可能多的field匹配的doc，而不是某个field完全匹配的doc --> 解决，要求每个term都必须在任何一个field中出现

Peter，Smith

要求Peter必须在author\_first\_name或author\_last\_name中出现

要求Smith必须在author\_first\_name或author\_last\_name中出现

Peter Smith可能是横跨在多个field中的，所以必须要求每个term都在某个field中出现，组合起来才能组成我们想要的标识，完整的人名

原来most\_fiels，可能像Smith Williams也可能会出现，因为most\_fields要求只是任何一个field匹配了就可以，匹配的field越多，分数越高

问题2：most\_fields，没办法用minimum\_should\_match去掉长尾数据，就是匹配的特别少的结果 --> 解决，既然每个term都要求出现，长尾肯定被去除掉了

java hadoop spark --> 这3个term都必须在任何一个field出现了

比如有的document，只有一个field中包含一个java，那就被干掉了，作为长尾就没了

问题3：TF/IDF算法，比如Peter Smith和Smith Williams，搜索Peter Smith的时候，由于first\_name中很少有Smith的，所以query在所有document中的频率很低，得到的分数很高，可能Smith Williams反而会排在Peter Smith前面 --> 计算IDF的时候，将每个query在每个field中的IDF都取出来，取最小值，就不会出现极端情况下的极大值了

Peter Smith

Peter

Smith

Smith，在author\_first\_name这个field中，在所有doc的这个Field中，出现的频率很低，导致IDF分数很高；Smith在所有doc的author\_last\_name field中的频率算出一个IDF分数，因为一般来说last\_name中的Smith频率都较高，所以IDF分数是正常的，不会太高；然后对于Smith来说，会取两个IDF分数中，较小的那个分数。就不会出现IDF分过高的情况。

## phrase matching（短语匹配）

* + - 1. 什么是近似匹配

两个句子

java is my favourite programming language, and I also think spark is a very good big data system.

java spark are very related, because scala is spark's programming language and scala is also based on jvm like java.

match query，搜索java spark

{

"match": {

"content": "java spark"

}

}

match query，只能搜索到包含java和spark的document，但是不知道java和spark是不是离的很近

包含java或包含spark，或包含java和spark的doc，都会被返回回来。我们其实并不知道哪个doc，java和spark距离的比较近。如果我们就是希望搜索java spark，中间不能插入任何其他的字符，那这个时候match去做全文检索，能搞定我们的需求吗？答案是，搞不定。

如果我们要尽量让java和spark离的很近的document优先返回，要给它一个更高的relevance score，这就涉及到了proximity match，近似匹配

如果说，要实现两个需求：

* 1. java spark，就靠在一起，中间不能插入任何其他字符，就要搜索出来这种doc
  2. java spark，但是要求，java和spark两个单词靠的越近，doc的分数越高，排名越靠前

要实现上述两个需求，用match做全文检索，是搞不定的，必须得用proximity match，近似匹配

phrase match：短语匹配

proximity match：近似匹配

这一讲，要学习的是phrase match，就是仅仅搜索出java和spark靠在一起的那些doc，比如有个doc，是java use'd spark，不行。必须是比如java spark are very good friends，是可以搜索出来的。

phrase match，就是要去将多个term作为一个短语，一起去搜索，只有包含这个短语的doc才会作为结果返回。不像是match，java spark，java的doc也会返回，spark的doc也会返回。

* + - 1. match\_phrase

GET /forum/article/\_search

{

"query": {

"match": {

"content": "java spark"

}

}

}

单单包含java的doc也返回了，不是我们想要的结果

POST /forum/article/5/\_update

{

"doc": {

"content": "spark is best big data solution based on scala ,an programming language similar to java spark"

}

}

将一个doc的content设置为恰巧包含java spark这个短语

match\_phrase语法

GET /forum/article/\_search

{

"query": {

"match\_phrase": {

"content": "java spark"

}

}

}

成功了，只有包含java spark这个短语的doc才返回了，只包含java的doc不会返回

* + - 1. term position

hello world, java spark doc1

hi, spark java doc2

hello doc1(0)

wolrd doc1(1)

java doc1(2) doc2(2)

spark doc1(3) doc2(1)

了解什么是分词后的position

GET \_analyze

{

"text": "hello world, java spark",

"analyzer": "standard"

}

* + - 1. match\_phrase的基本原理

索引中的position，match\_phrase

hello world, java spark doc1

hi, spark java doc2

hello doc1(0)

wolrd doc1(1)

java doc1(2) doc2(2)

spark doc1(3) doc2(1)

java spark --> match phrase

java spark --> java和spark

java --> doc1(2) doc2(2)

spark --> doc1(3) doc2(1)

要找到每个term都在的一个共有的那些doc，就是要求一个doc，必须包含每个term，才能拿出来继续计算

doc1 --> java和spark --> spark position恰巧比java大1 --> java的position是2，spark的position是3，恰好满足条件

doc1符合条件

doc2 --> java和spark --> java position是2，spark position是1，spark position比java position小1，而不是大1 --> 光是position就不满足，那么doc2不匹配

必须理解这块原理！！！！

因为后面的proximity match就是原理跟这个一模一样！！！

## slop参数实现近似匹配以及原理

GET /forum/article/\_search

{

"query": {

"match\_phrase": {

"title": {

"query": "java spark",

"slop": 1

}

}

}

}

slop的含义是什么？

query string，搜索文本，中的几个term，要经过几次移动才能与一个document匹配，这个移动的次数，就是slop

实际举例，一个query string经过几次移动之后可以匹配到一个document，然后设置slop

hello world, java is very good, spark is also very good.

java spark，match phrase，搜不到

如果我们指定了slop，那么就允许java spark进行移动，来尝试与doc进行匹配

java is very good spark is

java spark

java --> spark

java --> spark

java --> spark

这里的slop，就是3，因为java spark这个短语，spark移动了3次，就可以跟一个doc匹配上了

slop的含义，不仅仅是说一个query string terms移动几次，跟一个doc匹配上。一个query string terms，最多可以移动几次去尝试跟一个doc匹配上

slop，设置的是3，那么就ok

GET /forum/article/\_search

{

"query": {

"match\_phrase": {

"title": {

"query": "java spark",

"slop": 3

}

}

}

}

就可以把刚才那个doc匹配上，那个doc会作为结果返回

但是如果slop设置的是2，那么java spark，spark最多只能移动2次，此时跟doc是匹配不上的，那个doc是不会作为结果返回的

做实验，验证slop的含义

GET /forum/article/\_search

{

"query": {

"match\_phrase": {

"content": {

"query": "spark data",

"slop": 3

}

}

}

}

spark is best big data solution based on scala ,an programming language similar to java spark

spark data

--> data

--> data

spark --> data

GET /forum/article/\_search

{

"query": {

"match\_phrase": {

"content": {

"query": "data spark",

"slop": 5

}

}

}

}

spark is best big data

data spark

--> data/spark

spark <--data

spark --> data

spark --> data

spark --> data

slop搜索下，关键词离的越近，relevance score就会越高，做实验说明。。。

{

"took": 4,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 1.3728157,

"hits": [

{

"\_index": "forum",

"\_type": "article",

"\_id": "2",

"\_score": 1.3728157,

"\_source": {

"articleID": "KDKE-B-9947-#kL5",

"userID": 1,

"hidden": false,

"postDate": "2017-01-02",

"tag": [

"java"

],

"tag\_cnt": 1,

"view\_cnt": 50,

"title": "this is java blog",

"content": "i think java is the best programming language",

"sub\_title": "learned a lot of course",

"author\_first\_name": "Smith",

"author\_last\_name": "Williams",

"new\_author\_last\_name": "Williams",

"new\_author\_first\_name": "Smith"

}

},

{

"\_index": "forum",

"\_type": "article",

"\_id": "5",

"\_score": 0.5753642,

"\_source": {

"articleID": "DHJK-B-1395-#Ky5",

"userID": 3,

"hidden": false,

"postDate": "2017-03-01",

"tag": [

"elasticsearch"

],

"tag\_cnt": 1,

"view\_cnt": 10,

"title": "this is spark blog",

"content": "spark is best big data solution based on scala ,an programming language similar to java spark",

"sub\_title": "haha, hello world",

"author\_first\_name": "Tonny",

"author\_last\_name": "Peter Smith",

"new\_author\_last\_name": "Peter Smith",

"new\_author\_first\_name": "Tonny"

}

},

{

"\_index": "forum",

"\_type": "article",

"\_id": "1",

"\_score": 0.28582606,

"\_source": {

"articleID": "XHDK-A-1293-#fJ3",

"userID": 1,

"hidden": false,

"postDate": "2017-01-01",

"tag": [

"java",

"hadoop"

],

"tag\_cnt": 2,

"view\_cnt": 30,

"title": "this is java and elasticsearch blog",

"content": "i like to write best elasticsearch article",

"sub\_title": "learning more courses",

"author\_first\_name": "Peter",

"author\_last\_name": "Smith",

"new\_author\_last\_name": "Smith",

"new\_author\_first\_name": "Peter"

}

}

]

}

}

GET /forum/article/\_search

{

"query": {

"match\_phrase": {

"content": {

"query": "java best",

"slop": 15

}

}

}

}

{

"took": 3,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 2,

"max\_score": 0.65380025,

"hits": [

{

"\_index": "forum",

"\_type": "article",

"\_id": "2",

"\_score": 0.65380025,

"\_source": {

"articleID": "KDKE-B-9947-#kL5",

"userID": 1,

"hidden": false,

"postDate": "2017-01-02",

"tag": [

"java"

],

"tag\_cnt": 1,

"view\_cnt": 50,

"title": "this is java blog",

"content": "i think java is the best programming language",

"sub\_title": "learned a lot of course",

"author\_first\_name": "Smith",

"author\_last\_name": "Williams",

"new\_author\_last\_name": "Williams",

"new\_author\_first\_name": "Smith"

}

},

{

"\_index": "forum",

"\_type": "article",

"\_id": "5",

"\_score": 0.07111243,

"\_source": {

"articleID": "DHJK-B-1395-#Ky5",

"userID": 3,

"hidden": false,

"postDate": "2017-03-01",

"tag": [

"elasticsearch"

],

"tag\_cnt": 1,

"view\_cnt": 10,

"title": "this is spark blog",

"content": "spark is best big data solution based on scala ,an programming language similar to java spark",

"sub\_title": "haha, hello world",

"author\_first\_name": "Tonny",

"author\_last\_name": "Peter Smith",

"new\_author\_last\_name": "Peter Smith",

"new\_author\_first\_name": "Tonny"

}

}

]

}

}

其实，加了slop的phrase match，就是proximity match，近似匹配

1、java spark，短语，doc，phrase match

2、java spark，可以有一定的距离，但是靠的越近，越先搜索出来，proximity match

## 混合使用match和近似匹配实现召回率与精准度的平衡

召回率

比如你搜索一个java spark，总共有100个doc，能返回多少个doc作为结果，就是召回率，recall

精准度

比如你搜索一个java spark，能不能尽可能让包含java spark，或者是java和spark离的很近的doc，排在最前面，precision

直接用match\_phrase短语搜索，会导致必须所有term都在doc field中出现，而且距离在slop限定范围内，才能匹配上

match phrase，proximity match，要求doc必须包含所有的term，才能作为结果返回；如果某一个doc可能就是有某个term没有包含，那么就无法作为结果返回

java spark --> hello world java --> 就不能返回了

java spark --> hello world, java spark --> 才可以返回

近似匹配的时候，召回率比较低，精准度太高了

但是有时可能我们希望的是匹配到几个term中的部分，就可以作为结果出来，这样可以提高召回率。同时我们也希望用上match\_phrase根据距离提升分数的功能，让几个term距离越近分数就越高，优先返回

就是优先满足召回率，意思，java spark，包含java的也返回，包含spark的也返回，包含java和spark的也返回；同时兼顾精准度，就是包含java和spark，同时java和spark离的越近的doc排在最前面

此时可以用bool组合match query和match\_phrase query一起，来实现上述效果

GET /forum/article/\_search

{

"query": {

"bool": {

"must": {

"match": {

"title": {

"query": "java spark" --> java或spark或java spark，java和spark靠前，但是没法区分java和spark的距离，也许java和spark靠的很近，但是没法排在最前面

}

}

},

"should": {

"match\_phrase": { --> 在slop以内，如果java spark能匹配上一个doc，那么就会对doc贡献自己的relevance score，如果java和spark靠的越近，那么就分数越高

"title": {

"query": "java spark",

"slop": 50

}

}

}

}

}

}

GET /forum/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"content": "java spark"

}

}

]

}

}

}

{

"took": 5,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 2,

"max\_score": 0.68640786,

"hits": [

{

"\_index": "forum",

"\_type": "article",

"\_id": "2",

"\_score": 0.68640786,

"\_source": {

"articleID": "KDKE-B-9947-#kL5",

"userID": 1,

"hidden": false,

"postDate": "2017-01-02",

"tag": [

"java"

],

"tag\_cnt": 1,

"view\_cnt": 50,

"title": "this is java blog",

"content": "i think java is the best programming language",

"sub\_title": "learned a lot of course",

"author\_first\_name": "Smith",

"author\_last\_name": "Williams",

"new\_author\_last\_name": "Williams",

"new\_author\_first\_name": "Smith",

"followers": [

"Tom",

"Jack"

]

}

},

{

"\_index": "forum",

"\_type": "article",

"\_id": "5",

"\_score": 0.68324494,

"\_source": {

"articleID": "DHJK-B-1395-#Ky5",

"userID": 3,

"hidden": false,

"postDate": "2017-03-01",

"tag": [

"elasticsearch"

],

"tag\_cnt": 1,

"view\_cnt": 10,

"title": "this is spark blog",

"content": "spark is best big data solution based on scala ,an programming language similar to java spark",

"sub\_title": "haha, hello world",

"author\_first\_name": "Tonny",

"author\_last\_name": "Peter Smith",

"new\_author\_last\_name": "Peter Smith",

"new\_author\_first\_name": "Tonny",

"followers": [

"Jack",

"Robbin Li"

]

}

}

]

}

}

GET /forum/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"content": "java spark"

}

}

],

"should": [

{

"match\_phrase": {

"content": {

"query": "java spark",

"slop": 50

}

}

}

]

}

}

}

{

"took": 5,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 2,

"max\_score": 1.258609,

"hits": [

{

"\_index": "forum",

"\_type": "article",

"\_id": "5",

"\_score": 1.258609,

"\_source": {

"articleID": "DHJK-B-1395-#Ky5",

"userID": 3,

"hidden": false,

"postDate": "2017-03-01",

"tag": [

"elasticsearch"

],

"tag\_cnt": 1,

"view\_cnt": 10,

"title": "this is spark blog",

"content": "spark is best big data solution based on scala ,an programming language similar to java spark",

"sub\_title": "haha, hello world",

"author\_first\_name": "Tonny",

"author\_last\_name": "Peter Smith",

"new\_author\_last\_name": "Peter Smith",

"new\_author\_first\_name": "Tonny",

"followers": [

"Jack",

"Robbin Li"

]

}

},

{

"\_index": "forum",

"\_type": "article",

"\_id": "2",

"\_score": 0.68640786,

"\_source": {

"articleID": "KDKE-B-9947-#kL5",

"userID": 1,

"hidden": false,

"postDate": "2017-01-02",

"tag": [

"java"

],

"tag\_cnt": 1,

"view\_cnt": 50,

"title": "this is java blog",

"content": "i think java is the best programming language",

"sub\_title": "learned a lot of course",

"author\_first\_name": "Smith",

"author\_last\_name": "Williams",

"new\_author\_last\_name": "Williams",

"new\_author\_first\_name": "Smith",

"followers": [

"Tom",

"Jack"

]

}

}

]

}

}

## rescoring机制优化近似匹配搜索

match和phrase match(proximity match)区别

match --> 只要简单的匹配到了一个term，就可以理解将term对应的doc作为结果返回，扫描倒排索引，扫描到了就ok

phrase match --> 首先扫描到所有term的doc list; 找到包含所有term的doc list; 然后对每个doc都计算每个term的position，是否符合指定的范围; slop，需要进行复杂的运算，来判断能否通过slop移动，匹配一个doc

match query的性能比phrase match和proximity match（有slop）要高很多。因为后两者都要计算position的距离。

match query比phrase match的性能要高10倍，比proximity match的性能要高20倍。

但是别太担心，因为es的性能一般都在毫秒级别，match query一般就在几毫秒，或者几十毫秒，而phrase match和proximity match的性能在几十毫秒到几百毫秒之间，所以也是可以接受的。

优化proximity match的性能，一般就是减少要进行proximity match搜索的document数量。主要思路就是，用match query先过滤出需要的数据，然后再用proximity match来根据term距离提高doc的分数，同时proximity match只针对每个shard的分数排名前n个doc起作用，来重新调整它们的分数，这个过程称之为rescoring，重计分。因为一般用户会分页查询，只会看到前几页的数据，所以不需要对所有结果进行proximity match操作。

用我们刚才的说法，match + proximity match同时实现召回率和精准度

默认情况下，match也许匹配了1000个doc，proximity match全都需要对每个doc进行一遍运算，判断能否slop移动匹配上，然后去贡献自己的分数

但是很多情况下，match出来也许1000个doc，其实用户大部分情况下是分页查询的，所以可能最多只会看前几页，比如一页是10条，最多也许就看5页，就是50条

proximity match只要对前50个doc进行slop移动去匹配，去贡献自己的分数即可，不需要对全部1000个doc都去进行计算和贡献分数

rescore：重打分

match：1000个doc，其实这时候每个doc都有一个分数了; proximity match，前50个doc，进行rescore，重打分，即可; 让前50个doc，term举例越近的，排在越前面

GET /forum/article/\_search

{

"query": {

"match": {

"content": "java spark"

}

},

"rescore": {

"window\_size": 50,

"query": {

"rescore\_query": {

"match\_phrase": {

"content": {

"query": "java spark",

"slop": 50

}

}

}

}

}

}

## 前缀搜索、通配符搜索、正则搜索

1、前缀搜索

C3D0-KD345

C3K5-DFG65

C4I8-UI365

C3 --> 上面这两个都搜索出来 --> 根据字符串的前缀去搜索

不用帖子的案例背景，因为比较简单，直接用自己手动建的新索引，给大家演示一下就可以了

PUT my\_index

{

"mappings": {

"my\_type": {

"properties": {

"title": {

"type": "keyword"

}

}

}

}

}

GET my\_index/my\_type/\_search

{

"query": {

"prefix": {

"title": {

"value": "C3"

}

}

}

}

2、前缀搜索的原理

prefix query不计算relevance score，与prefix filter唯一的区别就是，filter会cache bitset

扫描整个倒排索引，举例说明

前缀越短，要处理的doc越多，性能越差，尽可能用长前缀搜索

前缀搜索，它是怎么执行的？性能为什么差呢？

match

C3-D0-KD345

C3-K5-DFG65

C4-I8-UI365

全文检索

每个字符串都需要被分词

c3 doc1,doc2

d0

kd345

k5

dfg65

c4

i8

ui365

c3 --> 扫描倒排索引 --> 一旦扫描到c3，就可以停了，因为带c3的就2个doc，已经找到了 --> 没有必要继续去搜索其他的term了

match性能往往是很高的

不分词

C3-D0-KD345

C3-K5-DFG65

C4-I8-UI365

c3 --> 先扫描到了C3-D0-KD345，很棒，找到了一个前缀带c3的字符串 --> 还是要继续搜索的，因为后面还有一个C3-K5-DFG65，也许还有其他很多的前缀带c3的字符串 --> 你扫描到了一个前缀匹配的term，不能停，必须继续搜索 --> 直到扫描完整个的倒排索引，才能结束

因为实际场景中，可能有些场景是全文检索解决不了的

C3D0-KD345

C3K5-DFG65

C4I8-UI365

c3d0

kd345

c3 --> match --> 扫描整个倒排索引，能找到吗

c3 --> 只能用prefix

prefix性能很差

3、通配符搜索

跟前缀搜索类似，功能更加强大

C3D0-KD345

C3K5-DFG65

C4I8-UI365

5字符-D任意个字符5

5?-\*5：通配符去表达更加复杂的模糊搜索的语义

GET my\_index/my\_type/\_search

{

"query": {

"wildcard": {

"title": {

"value": "C?K\*5"

}

}

}

}

?：任意字符

\*：0个或任意多个字符

性能一样差，必须扫描整个倒排索引，才ok

4、正则搜索

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

{

"query": {

"regexp": {

"title": "C[0-9].+"

}

}

}

C[0-9].+

[0-9]：指定范围内的数字

[a-z]：指定范围内的字母

.：一个字符

+：前面的正则表达式可以出现一次或多次

wildcard和regexp，与prefix原理一致，都会扫描整个索引，性能很差

主要是给大家介绍一些高级的搜索语法。在实际应用中，能不用尽量别用。性能太差了。

## match\_phrase\_prefix实现search-time搜索推荐

搜索推荐，search as you type，搜索提示，解释一下什么意思

hello w --> 搜索

hello world

hello we

hello win

hello wind

hello dog

hello cat

hello w -->

hello world

hello we

hello win

hello wind

搜索推荐的功能

百度 --> elas --> elasticsearch --> elasticsearch权威指南

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

{

"query": {

"match\_phrase\_prefix": {

"title": "hello d"

}

}

}

原理跟match\_phrase类似，唯一的区别，就是把最后一个term作为前缀去搜索

hello就是去进行match，搜索对应的doc

w，会作为前缀，去扫描整个倒排索引，找到所有w开头的doc

然后找到所有doc中，即包含hello，又包含w开头的字符的doc

根据你的slop去计算，看在slop范围内，能不能让hello w，正好跟doc中的hello和w开头的单词的position相匹配

也可以指定slop，但是只有最后一个term会作为前缀

max\_expansions：指定prefix最多匹配多少个term，超过这个数量就不继续匹配了，限定性能

默认情况下，前缀要扫描所有的倒排索引中的term，去查找w打头的单词，但是这样性能太差。可以用max\_expansions限定，w前缀最多匹配多少个term，就不再继续搜索倒排索引了。

尽量不要用，因为，最后一个前缀始终要去扫描大量的索引，性能可能会很差

## ngram分词机制实现index-time搜索推荐

1、ngram和index-time搜索推荐原理

什么是ngram

quick，5种长度下的ngram

ngram length=1，q u i c k

ngram length=2，qu ui ic ck

ngram length=3，qui uic ick

ngram length=4，quic uick

ngram length=5，quick

什么是edge ngram

quick，anchor首字母后进行ngram

q

qu

qui

quic

quick

使用edge ngram将每个单词都进行进一步的分词切分，用切分后的ngram来实现前缀搜索推荐功能

hello world

hello we

h

he

hel

hell

hello doc1,doc2

w doc1,doc2

wo

wor

worl

world

e doc2

helloworld

min ngram = 1

max ngram = 3

h

he

hel

hello w

hello --> hello，doc1

w --> w，doc1

doc1，hello和w，而且position也匹配，所以，ok，doc1返回，hello world

搜索的时候，不用再根据一个前缀，然后扫描整个倒排索引了; 简单的拿前缀去倒排索引中匹配即可，如果匹配上了，那么就好了; match，全文检索

2、实验一下ngram

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

"autocomplete\_filter": {

"type": "edge\_ngram",

"min\_gram": 1,

"max\_gram": 20

}

},

"analyzer": {

"autocomplete": {

"type": "custom",

"tokenizer": "standard",

"filter": [

"lowercase",

"autocomplete\_filter"

]

}

}

}

}

}

GET /my\_index/\_analyze

{

"analyzer": "autocomplete",

"text": "quick brown"

}

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

{

"properties": {

"title": {

"type": "string",

"analyzer": "autocomplete",

"search\_analyzer": "standard"

}

}

}

hello world

h

he

hel

hell

hello

w

wo

wor

worl

world

hello w

h

he

hel

hell

hello

w

hello w --> hello --> w

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

{

"query": {

"match\_phrase": {

"title": "hello w"

}

}

}

如果用match，只有hello的也会出来，全文检索，只是分数比较低

推荐使用match\_phrase，要求每个term都有，而且position刚好靠着1位，符合我们的期望的

## TF&IDF算法以及向量空间模型算法

1、boolean model

类似and这种逻辑操作符，先过滤出包含指定term的doc

query "hello world" --> 过滤 --> hello / world / hello & world

bool --> must/must not/should --> 过滤 --> 包含 / 不包含 / 可能包含

doc --> 不打分数 --> 正或反 true or false --> 为了减少后续要计算的doc的数量，提升性能

2、TF/IDF

单个term在doc中的分数

query: hello world --> doc.content

doc1: java is my favourite programming language, hello world !!!

doc2: hello java, you are very good, oh hello world!!!

hello对doc1的评分

TF: term frequency

找到hello在doc1中出现了几次，1次，会根据出现的次数给个分数

一个term在一个doc中，出现的次数越多，那么最后给的相关度评分就会越高

IDF：inversed document frequency

找到hello在所有的doc中出现的次数，3次

一个term在所有的doc中，出现的次数越多，那么最后给的相关度评分就会越低

length norm

hello搜索的那个field的长度，field长度越长，给的相关度评分越低; field长度越短，给的相关度评分越高

最后，会将hello这个term，对doc1的分数，综合TF，IDF，length norm，计算出来一个综合性的分数

hello world --> doc1 --> hello对doc1的分数，world对doc1的分数 --> 但是最后hello world query要对doc1有一个总的分数 --> vector space model

3、vector space model

多个term对一个doc的总分数

hello world --> es会根据hello world在所有doc中的评分情况，计算出一个query vector，query向量

hello这个term，给的基于所有doc的一个评分就是2

world这个term，给的基于所有doc的一个评分就是5

[2, 5]

query vector

doc vector，3个doc，一个包含1个term，一个包含另一个term，一个包含2个term

3个doc

doc1：包含hello --> [2, 0]

doc2：包含world --> [0, 5]

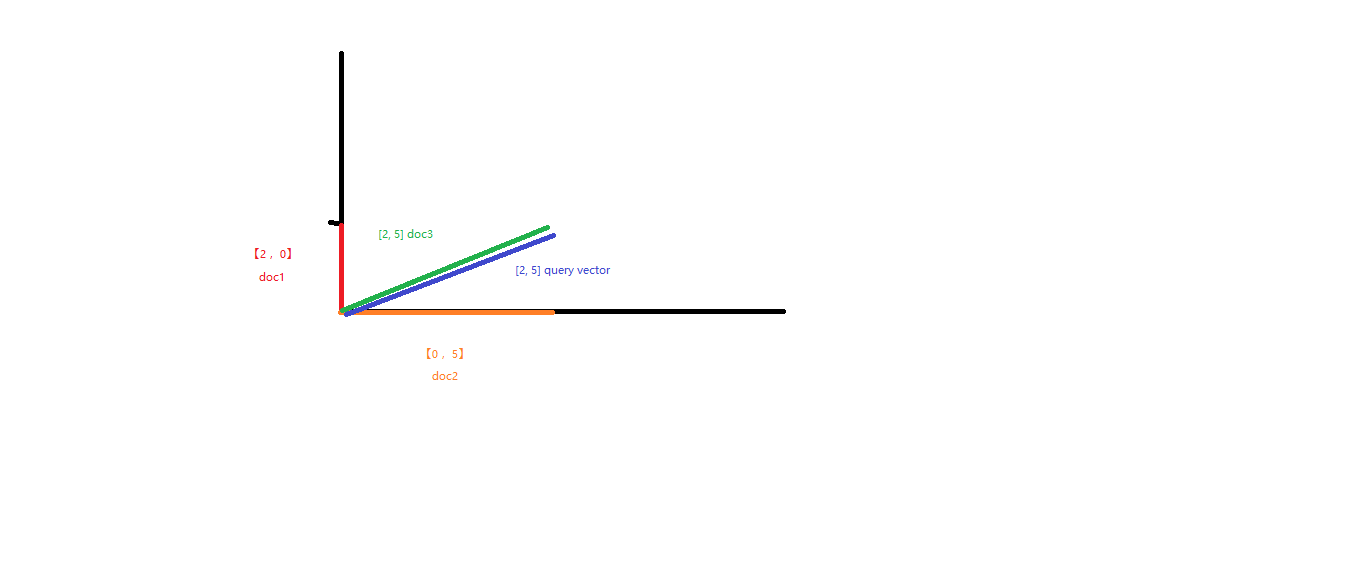
doc3：包含hello, world --> [2, 5]

会给每一个doc，拿每个term计算出一个分数来，hello有一个分数，world有一个分数，再拿所有term的分数组成一个doc vector

画在一个图中，取每个doc vector对query vector的弧度，给出每个doc对多个term的总分数

每个doc vector计算出对query vector的弧度，最后基于这个弧度给出一个doc相对于query中多个term的总分数

弧度越大，分数月底; 弧度越小，分数越高



如果是多个term，那么就是线性代数来计算，无法用图表示

**空间向量模型**

## lucene的相关度分数算法

boolean model、TF/IDF、vector space model

深入讲解TF/IDF算法，在lucene中，底层，到底进行TF/IDF算法计算的一个完整的公式是什么？

0、boolean model

query: hello world

"match": {

"title": "hello world"

}

"bool": {

"should": [

{

"match": {

"title": "hello"

}

},

{

"natch": {

"title": "world"

}

}

]

}

普通multivalue搜索，转换为bool搜索，boolean model

* 1. lucene practical scoring function

practical scoring function，来计算一个query对一个doc的分数的公式，该函数会使用一个公式来计算

score(q,d) =

queryNorm(q)

· coord(q,d)

· ∑ (

tf(t in d)

· idf(t)2

· t.getBoost()

· norm(t,d)

) (t in q)

score(q,d) score(q,d) is the relevance score of document d for query q.

这个公式的最终结果，就是说是一个query（叫做q），对一个doc（叫做d）的最终的总评分

queryNorm(q) is the query normalization factor (new).

queryNorm，是用来让一个doc的分数处于一个合理的区间内，不要太离谱，举个例子，一个doc分数是10000，一个doc分数是0.1，你们说好不好，肯定不好

coord(q,d) is the coordination factor (new).

简单来说，就是对更加匹配的doc，进行一些分数上的成倍的奖励

The sum of the weights for each term t in the query q for document d.

∑：求和的符号

∑ (t in q)：query中每个term，query = hello world，query中的term就包含了hello和world

query中每个term对doc的分数，进行求和，多个term对一个doc的分数，组成一个vector space，然后计算吗，就在这一步

tf(t in d) is the term frequency for term t in document d.

计算每一个term对doc的分数的时候，就是TF/IDF算法

idf(t) is the inverse document frequency for term t.

t.getBoost() is the boost that has been applied to the query (new).

norm(t,d) is the field-length norm, combined with the index-time field-level boost, if any. (new).

* 1. query normalization factor

queryNorm = 1 / √sumOfSquaredWeights

sumOfSquaredWeights = 所有term的IDF分数之和，开一个平方根，然后做一个平方根分之1

主要是为了将分数进行规范化 --> 开平方根，首先数据就变小了 --> 然后还用1去除以这个平方根，分数就会很小 --> 1.几 / 零点几

分数就不会出现几万，几十万，那样的离谱的分数

* 1. query coodination

奖励那些匹配更多字符的doc更多的分数

Document 1 with hello → score: 1.5

Document 2 with hello world → score: 3.0

Document 3 with hello world java → score: 4.5

Document 1 with hello → score: 1.5 \* 1 / 3 = 0.5

Document 2 with hello world → score: 3.0 \* 2 / 3 = 2.0

Document 3 with hello world java → score: 4.5 \* 3 / 3 = 4.5

把计算出来的总分数 \* 匹配上的term数量 / 总的term数量，让匹配不同term/query数量的doc，分数之间拉开差距

* 1. field level boost

## 四种常见的相关度分数优化

对相关度评分进行调节和优化的常见的4种方法

1. query-time boost

GET /forum/article/\_search

{

"query": {

"bool": {

"should": [

{

"match": {

"title": {

"query": "java spark",

"boost": 2

}

}

},

{

"match": {

"content": "java spark"

}

}

]

}

}

}

1. 重构查询结构

重构查询结果，在es新版本中，影响越来越小了。一般情况下，没什么必要的话，大家不用也行。

GET /forum/article/\_search

{

"query": {

"bool": {

"should": [

{

"match": {

"content": "java"

}

},

{

"match": {

"content": "spark"

}

},

{

"bool": {

"should": [

{

"match": {

"content": "solution"

}

},

{

"match": {

"content": "beginner"

}

}

]

}

}

]

}

}

}

1. negative boost

搜索包含java，不包含spark的doc，但是这样子很死板

搜索包含java，尽量不包含spark的doc，如果包含了spark，不会说排除掉这个doc，而是说将这个doc的分数降低

包含了negative term的doc，分数乘以negative boost，分数降低

boosting query:一种复合查询，分为positive子查询和negitive子查询，两者的查询结构都会返回。positive子查询的score保持不变，negetive子查询的值将会根据negative\_boost的值做相应程度的降低。

GET /forum/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"content": "java"

}

}

],

"must\_not": [

{

"match": {

"content": "spark"

}

}

]

}

}

}

GET /forum/article/\_search

{

"query": {

"boosting": {

"positive": {

"match": {

"content": "java"

}

},

"negative": {

"match": {

"content": "spark"

}

},

"negative\_boost": 0.2

}

}

}

negative的doc，会乘以negative\_boost，降低分数

1. constant\_score

如果你压根儿不需要相关度评分，直接走constant\_score加filter，所有的doc分数都是1，没有评分的概念了

GET /forum/article/\_search

{

"query": {

"bool": {

"should": [

{

"constant\_score": {

"query": {

"match": {

"title": "java"

}

}

}

},

{

"constant\_score": {

"query": {

"match": {

"title": "spark"

}

}

}

}

]

}

}

}

## function\_score自定义相关度分数算法

可以做到自定义一个function\_score函数，自己将某个field的值，跟es内置算出来的分数进行运算，然后由自己指定的field来进行分数的增强

给所有的帖子数据增加follower数量

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"follower\_num" : 5} }

{ "update": { "\_id": "2"} }

{ "doc" : {"follower\_num" : 10} }

{ "update": { "\_id": "3"} }

{ "doc" : {"follower\_num" : 25} }

{ "update": { "\_id": "4"} }

{ "doc" : {"follower\_num" : 3} }

{ "update": { "\_id": "5"} }

{ "doc" : {"follower\_num" : 60} }

将对帖子搜索得到的分数，跟follower\_num进行运算，由follower\_num在一定程度上增强帖子的分数

看帖子的人越多，那么帖子的分数就越高

GET /forum/article/\_search

{

"query": {

"function\_score": {

"query": {

"multi\_match": {

"query": "java spark",

"fields": ["tile", "content"]

}

},

"field\_value\_factor": {

"field": "follower\_num",

"modifier": "log1p",

"factor": 0.5

},

"boost\_mode": "sum",

"max\_boost": 2

}

}

}

如果只有field，那么会将每个doc的分数都乘以follower\_num，如果有的doc follower是0，那么分数就会变为0，效果很不好。因此一般会加个log1p函数，公式会变为，new\_score = old\_score \* log(1 + number\_of\_votes)，这样出来的分数会比较合理

再加个factor，可以进一步影响分数，new\_score = old\_score \* log(1 + factor \* number\_of\_votes)

boost\_mode，可以决定分数与指定字段的值如何计算，multiply，sum，min，max，replace

max\_boost，限制计算出来的分数不要超过max\_boost指定的值

## 误拼写时的fuzzy模糊搜索

搜索的时候，可能输入的搜索文本会出现误拼写的情况

doc1: hello world

doc2: hello java

搜索：hallo world

fuzzy搜索技术 --> 自动将拼写错误的搜索文本，进行纠正，纠正以后去尝试匹配索引中的数据

POST /my\_index/my\_type/\_bulk

{ "index": { "\_id": 1 }}

{ "text": "Surprise me!"}

{ "index": { "\_id": 2 }}

{ "text": "That was surprising."}

{ "index": { "\_id": 3 }}

{ "text": "I wasn't surprised."}

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

{

"query": {

"fuzzy": {

"text": {

"value": "surprize",

"fuzziness": 2

}

}

}

}

surprize --> 拼写错误 --> surprise --> s -> z

surprize --> surprise -> z -> s，纠正一个字母，就可以匹配上，所以在fuziness指定的2范围内

surprize --> surprised -> z -> s，末尾加个d，纠正了2次，也可以匹配上，在fuziness指定的2范围内

surprize --> surprising -> z -> s，去掉e，ing，3次，总共要5次，才可以匹配上，始终纠正不了

fuzzy搜索以后，会自动尝试将你的搜索文本进行纠错，然后去跟文本进行匹配

fuzziness，你的搜索文本最多可以纠正几个字母去跟你的数据进行匹配，默认如果不设置，就是2

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

{

"query": {

"match": {

"text": {

"query": "SURPIZE ME",

"fuzziness": "AUTO",

"operator": "and"

}

}

}

}

## IK中文分词器的安装和使用

standard：没有办法对中文进行合理分词的，只是将每个中文字符一个一个的切割开来，比如说中国人 --> 中 国 人

英语的也要学：所以说，我们利用核心知识篇的相关的知识，来把es这种英文原生的搜索引擎，先学一下; 因为有些知识点，可能用英文讲更靠谱，因为比如说analyzed，palyed，students --> stemmer，analyze，play，student。有些知识点，仅仅适用于英文，不太适用于中文

从这一讲开始，大家就会觉得很爽，因为全部都是我们熟悉的中文了，没有英文了，高阶知识点，搜索，聚合，全部是中文了

在搜索引擎领域，比较成熟和流行的，就是ik分词器

中国人很喜欢吃油条

standard：中 国 人 很 喜 欢 吃 油 条

ik：中国人 很 喜欢 吃 油条

* 1. 在elasticsearch中安装ik中文分词器

1. git clone <https://github.com/medcl/elasticsearch-analys>is-ik
2. git checkout tags/v5.2.0
3. mvn package
4. 将target/releases/elasticsearch-analysis-ik-5.2.0

.zip拷贝到es/plugins/ik目录下

1. 在es/plugins/ik下对elasticsearch-analysis-ik-5.2.0.zip进行解压缩
2. 重启es
   1. ik分词器基础知识

两种analyzer，你根据自己的需要自己选吧，但是一般是选用ik\_max\_word

ik\_max\_word: 会将文本做最细粒度的拆分，比如会将“中华人民共和国国歌”拆分为“中华人民共和国,中华人民,中华,华人,人民共和国,人民,人,民,共和国,共和,和,国国,国歌”，会穷尽各种可能的组合；

ik\_smart: 会做最粗粒度的拆分，比如会将“中华人民共和国国歌”拆分为“中华人民共和国,国歌”。

共和国 --> 中华人民共和国和国歌，搜到吗？？？？

* 1. ik分词器的使用

PUT /my\_index

{

"mappings": {

"my\_type": {

"properties": {

"text": {

"type": "text",

"analyzer": "ik\_max\_word"

}

}

}

}

}

POST /my\_index/my\_type/\_bulk

{ "index": { "\_id": "1"} }

{ "text": "男子偷上万元发红包求交女友 被抓获时仍然单身" }

{ "index": { "\_id": "2"} }

{ "text": "16岁少女为结婚“变”22岁 7年后想离婚被法院拒绝" }

{ "index": { "\_id": "3"} }

{ "text": "深圳女孩骑车逆行撞奔驰 遭索赔被吓哭(图)" }

{ "index": { "\_id": "4"} }

{ "text": "女人对护肤品比对男票好？网友神怼" }

{ "index": { "\_id": "5"} }

{ "text": "为什么国内的街道招牌用的都是红黄配？" }

GET /my\_index/\_analyze

{

"text": "男子偷上万元发红包求交女友 被抓获时仍然单身",

"analyzer": "ik\_max\_word"

}

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

{

"query": {

"match": {

"text": "16岁少女结婚好还是单身好？"

}

}

}

## IK分词器配置文件讲解以及自定义词库

* 1. ik配置文件

ik配置文件地址：es/plugins/ik/config目录

IKAnalyzer.cfg.xml：用来配置自定义词库

main.dic：ik原生内置的中文词库，总共有27万多条，只要是这些单词，

都会被分在一起

quantifier.dic：放了一些单位相关的词

suffix.dic：放了一些后缀

surname.dic：中国的姓氏

stopword.dic：英文停用词

ik原生最重要的两个配置文件

main.dic：包含了原生的中文词语，会按照这个里面的词语去分词

stopword.dic：包含了英文的停用词

停用词，stopword

a the and at but

一般，像停用词，会在分词的时候，直接被干掉，不会建立在倒排索引中

* 1. 自定义词库

1. 自己建立词库：每年都会涌现一些特殊的流行词，网红，蓝瘦香菇，喊麦，鬼畜，一般不会在ik的原生词典里

自己补充自己的最新的词语，到ik的词库里面去

IKAnalyzer.cfg.xml：ext\_dict，custom/mydict.dic

补充自己的词语，然后需要重启es，才能生效

1. 自己建立停用词库：比如了，的，啥，么，我们可能并不想去建立索引，让人家搜索

custom/ext\_stopword.dic，已经有了常用的中文停用词，可以补充自己的停用词，然后重启es

## 修改IK分词器源码来基于mysql热更新词库

es不停机，直接我们在外部某个地方添加新的词语，es中立即热加载到这些新词语

热更新的方案

1. 修改ik分词器源码，然后手动支持从mysql中每隔一定时间，自动加载新的词库
2. 基于ik分词器原生支持的热更新方案，部署一个web服务器，提供一个http接口，通过modified和tag两个http响应头，来提供词语的热更新

用第一种方案，第二种，ik git社区官方都不建议采用，觉得不太稳定

1. 下载源码
2. https://github.com/medcl/elasticsearch-analysis-ik/tree/v5.2.0
3. ik分词器，是个标准的java maven工程，直接导入eclipse就可以看到源码

修改源码

1. Dictionary类，169行：Dictionary单例类的初始化方法，在这里需要创建一个我们自定义的线程，并且启动它
2. HotDictReloadThread类：就是死循环，不断调用Dictionary.getSingleton().reLoadMainDict()，去重新加载词典
3. Dictionary类，389行：this.loadMySQLExtDict();
4. Dictionary类，683行：this.loadMySQLStopwordDict();

mvn package打包代码

1. target\releases\elasticsearch-analysis-ik-5.2.0.zip

解压缩ik压缩包

1. 将mysql驱动jar，放入ik的目录下

修改jdbc相关配置

重启es

1. 观察日志，日志中就会显示我们打印的那些东西，比如加载了什么配置，加载了什么词语，什么停用词

在mysql中添加词库与停用词

## bucket与metric两个核心概念

* 1. 文本编辑器介绍
  2. windows操作系统，原生的txt文本编辑器，一些json格式，不太方便去调整
  3. notepad++，功能不是太丰富
  4. sublime，整个功能也比较丰富，比较好，自己可以上网去下载，官网，免费的
  5. 两个核心概念：bucket和metric

bucket：一个数据分组

city name

北京 小李

北京 小王

上海 小张

上海 小丽

上海 小陈

基于city划分buckets

划分出来两个bucket，一个是北京bucket，一个是上海bucket

北京bucket：包含了2个人，小李，小王

上海bucket：包含了3个人，小张，小丽，小陈

按照某个字段进行bucket划分，那个字段的值相同的那些数据，就会被划分到一个bucket中

有一些mysql的sql知识的话，聚合，首先第一步就是分组，对每个组内的数据进行聚合分析，分组，就是我们的bucket

metric：对一个数据分组执行的统计

当我们有了一堆bucket之后，就可以对每个bucket中的数据进行聚合分词了，比如说计算一个bucket内所有数据的数量，或者计算一个bucket内所有数据的平均值，最大值，最小值

metric，就是对一个bucket执行的某种聚合分析的操作，比如说求平均值，求最大值，求最小值

select count(\*)

from access\_log

group by user\_id

bucket：group by user\_id --> 那些user\_id相同的数据，就会被划分到一个bucket中

metric：count(\*)，对每个user\_id bucket中所有的数据，计算一个数量

## 家电卖场案例以及统计哪种颜色电视销量最高

1. 家电卖场案例

以一个家电卖场中的电视销售数据为背景，来对各种品牌，各种颜色的电视的销量和销售额，进行各种各样角度的分析

PUT /tvs

{

"mappings": {

"sales": {

"properties": {

"price": {

"type": "long"

},

"color": {

"type": "keyword"

},

"brand": {

"type": "keyword"

},

"sold\_date": {

"type": "date"

}

}

}

}

}

POST /tvs/sales/\_bulk

{ "index": {}}

{ "price" : 1000, "color" : "红色", "brand" : "长虹", "sold\_date" : "2016-10-28" }

{ "index": {}}

{ "price" : 2000, "color" : "红色", "brand" : "长虹", "sold\_date" : "2016-11-05" }

{ "index": {}}

{ "price" : 3000, "color" : "绿色", "brand" : "小米", "sold\_date" : "2016-05-18" }

{ "index": {}}

{ "price" : 1500, "color" : "蓝色", "brand" : "TCL", "sold\_date" : "2016-07-02" }

{ "index": {}}

{ "price" : 1200, "color" : "绿色", "brand" : "TCL", "sold\_date" : "2016-08-19" }

{ "index": {}}

{ "price" : 2000, "color" : "红色", "brand" : "长虹", "sold\_date" : "2016-11-05" }

{ "index": {}}

{ "price" : 8000, "color" : "红色", "brand" : "三星", "sold\_date" : "2017-01-01" }

{ "index": {}}

{ "price" : 2500, "color" : "蓝色", "brand" : "小米", "sold\_date" : "2017-02-12" }

1. 统计哪种颜色的电视销量最高

GET /tvs/sales/\_search

{

"size" : 0,

"aggs" : {

"popular\_colors" : {

"terms" : {

"field" : "color"

}

}

}

}

size：只获取聚合结果，而不要执行聚合的原始数据

aggs：固定语法，要对一份数据执行分组聚合操作

popular\_colors：就是对每个aggs，都要起一个名字，这个名字是随机的，你随便取什么都ok

terms：根据字段的值进行分组

field：根据指定的字段的值进行分组

{

"took": 61,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 8,

"max\_score": 0,

"hits": []

},

"aggregations": {

"popular\_color": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "红色",

"doc\_count": 4

},

{

"key": "绿色",

"doc\_count": 2

},

{

"key": "蓝色",

"doc\_count": 2

}

]

}

}

}

hits.hits：我们指定了size是0，所以hits.hits就是空的，否则会把执

行聚合的那些原始数据给你返回回来

aggregations：聚合结果

popular\_color：我们指定的某个聚合的名称

buckets：根据我们指定的field划分出的buckets

key：每个bucket对应的那个值

doc\_count：这个bucket分组内，有多少个数据数量，其实就是这种颜色的销

量

## bucket+metric：统计每种颜色电视平均价格

GET /tvs/sales/\_search

{

"size" : 0,

"aggs": {

"colors": {

"terms": {

"field": "color"

},

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

按照color去分bucket，可以拿到每个color bucket中的数量，这个仅仅只是一个bucket操作，doc\_count其实只是es的bucket操作默认执行的一个内置metric

这一讲，就是除了bucket操作，分组，还要对每个bucket执行一个metric聚合统计操作

在一个aggs执行的bucket操作（terms），平级的json结构下，再加一个aggs，这个第二个aggs内部，同样取个名字，执行一个metric操作，avg，对之前的每个bucket中的数据的指定的field，price field，求一个平均值

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

就是一个metric，就是一个对一个bucket分组操作之后，对每个bucket都要执行的一个metric

第一个metric，avg，求指定字段的平均值

{

"took": 28,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 8,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_color": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "红色",

"doc\_count": 4,

"avg\_price": {

"value": 3250

}

},

{

"key": "绿色",

"doc\_count": 2,

"avg\_price": {

"value": 2100

}

},

{

"key": "蓝色",

"doc\_count": 2,

"avg\_price": {

"value": 2000

}

}

]

}

}

}

buckets，除了key和doc\_count

avg\_price：我们自己取的metric aggs的名字

value：我们的metric计算的结果，每个bucket中的数据的price字段求平

均值后的结果

select avg(price)

from tvs.sales

group by color

## bucket嵌套实现颜色+品牌的多层下钻分析

从颜色到品牌进行下钻分析，每种颜色的平均价格，以及找到每种颜色每个品牌的平均价格

我们可以进行多层次的下钻

比如说，现在红色的电视有4台，同时这4台电视中，有3台是属于长虹的，1台是属于小米的

红色电视中的3台长虹的平均价格是多少？

红色电视中的1台小米的平均价格是多少？

下钻的意思是，已经分了一个组了，比如说颜色的分组，然后还要继续对这个分组内的数据，再分组，比如一个颜色内，还可以分成多个不同的品牌的组，最后对每个最小粒度的分组执行聚合分析操作，这就叫做下钻分析

es，下钻分析，就要对bucket进行多层嵌套，多次分组

按照多个维度（颜色+品牌）多层下钻分析，而且学会了每个下钻维度（颜色，颜色+品牌），都可以对每个维度分别执行一次metric聚合操作

GET /tvs/sales/\_search

{

"size": 0,

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color"

},

"aggs": {

"color\_avg\_price": {

"avg": {

"field": "price"

}

},

"group\_by\_brand": {

"terms": {

"field": "brand"

},

"aggs": {

"brand\_avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

}

}

{

"took": 8,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 8,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_color": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "红色",

"doc\_count": 4,

"color\_avg\_price": {

"value": 3250

},

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "长虹",

"doc\_count": 3,

"brand\_avg\_price": {

"value": 1666.6666666666667

}

},

{

"key": "三星",

"doc\_count": 1,

"brand\_avg\_price": {

"value": 8000

}

}

]

}

},

{

"key": "绿色",

"doc\_count": 2,

"color\_avg\_price": {

"value": 2100

},

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "TCL",

"doc\_count": 1,

"brand\_avg\_price": {

"value": 1200

}

},

{

"key": "小米",

"doc\_count": 1,

"brand\_avg\_price": {

"value": 3000

}

}

]

}

},

{

"key": "蓝色",

"doc\_count": 2,

"color\_avg\_price": {

"value": 2000

},

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "TCL",

"doc\_count": 1,

"brand\_avg\_price": {

"value": 1500

}

},

{

"key": "小米",

"doc\_count": 1,

"brand\_avg\_price": {

"value": 2500

}

}

]

}

}

]

}

}

}

## 更多metrics：统计每种颜色电视最大最小价格

要学更多的metric

count，avg

count：bucket，terms，自动就会有一个doc\_count，就相当于是count

avg：avg aggs，求平均值

max：求一个bucket内，指定field值最大的那个数据

min：求一个bucket内，指定field值最小的那个数据

sum：求一个bucket内，指定field值的总和

一般来说，90%的常见的数据分析的操作，metric，无非就是count，avg，max，min，sum

m

GET /tvs/sales/\_search

{

"size" : 0,

"aggs": {

"colors": {

"terms": {

"field": "color"

},

"aggs": {

"avg\_price": { "avg": { "field": "price" } },

"min\_price" : { "min": { "field": "price"} },

"max\_price" : { "max": { "field": "price"} },

"sum\_price" : { "sum": { "field": "price" } }

}

}

}

}

求总和，就可以拿到一个颜色下的所有电视的销售总额

{

"took": 16,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 8,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_color": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "红色",

"doc\_count": 4,

"max\_price": {

"value": 8000

},

"min\_price": {

"value": 1000

},

"avg\_price": {

"value": 3250

},

"sum\_price": {

"value": 13000

}

},

{

"key": "绿色",

"doc\_count": 2,

"max\_price": {

"value": 3000

},

"min\_price": {

"value":

}, 1200

"avg\_price": {

"value": 2100

},

"sum\_price": {

"value": 4200

}

},

{

"key": "蓝色",

"doc\_count": 2,

"max\_price": {

"value": 2500

},

"min\_price": {

"value": 1500

},

"avg\_price": {

"value": 2000

},

"sum\_price": {

"value": 4000

}

}

]

}

}

}

## hitogram按价格区间统计电视销量和销售额

histogram：类似于terms，也是进行bucket分组操作，接收一个field，按照这个field的值的各个范围区间，进行bucket分组操作

"histogram":{

"field": "price",

"interval": 2000

},

interval：2000，划分范围，0~2000，2000~4000，4000~6000，6000~8000，8000~10000，buckets

去根据price的值，比如2500，看落在哪个区间内，比如2000~4000，此时就会将这条数据放入2000~4000对应的那个bucket中

bucket划分的方法，terms，将field值相同的数据划分到一个bucket中

bucket有了之后，一样的，去对每个bucket执行avg，count，sum，max，min，等各种metric操作，聚合分析

GET /tvs/sales/\_search

{

"size" : 0,

"aggs":{

"price":{

"histogram":{

"field": "price",

"interval": 2000

},

"aggs":{

"revenue": {

"sum": {

"field" : "price"

}

}

}

}

}

}

{

"took": 13,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 8,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_price": {

"buckets": [

{

"key": 0,

"doc\_count": 3,

"sum\_price": {

"value": 3700

}

},

{

"key": 2000,

"doc\_count": 4,

"sum\_price": {

"value": 9500

}

},

{

"key": 4000,

"doc\_count": 0,

"sum\_price": {

"value": 0

}

},

{

"key": 6000,

"doc\_count: {

"value":": 0,

"sum\_price" 0

}

},

{

"key": 8000,

"doc\_count": 1,

"sum\_price": {

"value": 8000

}

}

]

}

}

}

## date hitogram之统计每月电视销量

bucket，分组操作，histogram，按照某个值指定的interval，划分一个一个的bucket

date histogram，按照我们指定的某个date类型的日期field，以及日期interval，按照一定的日期间隔，去划分bucket

date interval = 1m，

2017-01-01~2017-01-31，就是一个bucket

2017-02-01~2017-02-28，就是一个bucket

然后会去扫描每个数据的date field，判断date落在哪个bucket中，就将其放入那个bucket

2017-01-05，就将其放入2017-01-01~2017-01-31，就是一个bucket

min\_doc\_count：即使某个日期interval，2017-01-01~2017-01-31中，一条数据都没有，那么这个区间也是要返回的，不然默认是会过滤掉这个区间的

extended\_bounds，min，max：划分bucket的时候，会限定在这个起始日期，和截止日期内

GET /tvs/sales/\_search

{

"size" : 0,

"aggs": {

"sales": {

"date\_histogram": {

"field": "sold\_date",

"interval": "month",

"format": "yyyy-MM-dd",

"min\_doc\_count" : 0,

"extended\_bounds" : {

"min" : "2016-01-01",

"max" : "2017-12-31"

}

}

}

}

}

{

"took": 16,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 8,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_sold\_date": {

"buckets": [

{

"key\_as\_string": "2016-01-01",

"key": 1451606400000,

"doc\_count": 0

},

{

"key\_as\_string": "2016-02-01",

"key": 1454284800000,

"doc\_count": 0

},

{

"key\_as\_string": "2016-03-01",

"key": 1456790400000,

"doc\_count": 0

},

{

"key\_as\_string": "2016-04-01",

"key": 1459468800000,

"doc\_count": 0

},

{

"key\_as\_string": "2016-05-01",

"key": 1462060800000,

"doc\_count": 1

},

{

"key\_as\_string": "2016-06-01",

"key": 1464739200000,

"doc\_count": 0

},

{

"key\_as\_string": "2016-07-01",

"key": 1467331200000,

"doc\_count": 1

},

{

"key\_as\_strin

"key\_as\_string": "2016-09-01",

"key": 1472688000000,

"doc\_count": 0

},g": "2016-08-01",

"key": 1470009600000,

"doc\_count": 1

},

{

{

"key\_as\_string": "2016-10-01",

"key": 1475280000000,

"doc\_count": 1

},

{

"key\_as\_string": "2016-11-01",

"key": 1477958400000,

"doc\_count": 2

},

{

"key\_as\_string": "2016-12-01",

"key": 1480550400000,

"doc\_count": 0

},

{

"key\_as\_string": "2017-01-01",

"key": 1483228800000,

"doc\_count": 1

},

{

"key\_as\_string": "2017-02-01",

"key": 1485907200000,

"doc\_count": 1

}

]

}

}

}

## 统计每季度每个品牌的销售额

GET /tvs/sales/\_search

{

"size": 0,

"aggs": {

"group\_by\_sold\_date": {

"date\_histogram": {

"field": "sold\_date",

"interval": "quarter",

"format": "yyyy-MM-dd",

"min\_doc\_count": 0,

"extended\_bounds": {

"min": "2016-01-01",

"max": "2017-12-31"

}

},

"aggs": {

"group\_by\_brand": {

"terms": {

"field": "brand"

},

"aggs": {

"sum\_price": {

"sum": {

"field": "price"

}

}

}

},

"total\_sum\_price": {

"sum": {

"field": "price"

}

}

}

}

}

}

{

"took": 10,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 8,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_sold\_date": {

"buckets": [

{

"key\_as\_string": "2016-01-01",

"key": 1451606400000,

"doc\_count": 0,

"total\_sum\_price": {

"value": 0

},

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": []

}

},

{

"key\_as\_string": "2016-04-01",

"key": 1459468800000,

"doc\_count": 1,

"total\_sum\_price": {

"value": 3000

},

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "小米",

"doc\_count": 1,

"sum\_price": {

"value": 3000

}

}

]

}

},

{

"key\_as\_string": "2016-07-01",

"key": 1467331200000,

"doc\_count": 2,

"total\_sum\_price": {

"value": 2700

},

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "TCL",

"doc\_count": 2,

"sum\_price": {

"value": 2700

}

}

]

}

},

{

"key\_as\_string": "2016-10-01",

"key": 1475280000000,

"doc\_count": 3,

"total\_sum\_price": {

"value": 5000

},

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "长虹",

"doc\_count": 3,

"sum\_price": {

"value": 5000

}

}

]

}

},

{

"key\_as\_string": "2017-01-01",

"key": 1483228800000,

"doc\_count": 2,

"total\_sum\_price": {

"value": 10500

},

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "三星",

"doc\_count": 1,

"sum\_price": {

"value": 8000

}

},

{

"key": "小米",

"doc\_count": 1,

"sum\_price": {

"value": 2500

}

}

]

}

},

{

"key\_as\_string": "2017-04-01",

"key": 1491004800000,

"doc\_count": 0,

"total\_sum\_price": {

"value": 0

},

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": []

}

},

{

"key\_as\_string": "2017-07-01",

"key": 1498867200000,

"doc\_count": 0,

"total\_sum\_price": {

"value": 0

},

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": []

}

},

{

"key\_as\_string": "2017-10-01",

"key": 1506816000000,

"doc\_count": 0,

"total\_sum\_price": {

"value": 0

},

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": []

}

}

]

}

}

}

## 统计指定品牌下每个颜色的销量

实际上来说，我们之前学习的搜索相关的知识，完全可以和聚合组合起来使用

select count(\*)

from tvs.sales

where brand like "%长%"

group by price

es aggregation，scope，任何的聚合，都必须在搜索出来的结果数据中之行，搜索结果，就是聚合分析操作的scope

GET /tvs/sales/\_search

{

"size": 0,

"query": {

"term": {

"brand": {

"value": "小米"

}

}

},

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color"

}

}

}

}

{

"took": 5,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 2,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_color": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "绿色",

"doc\_count": 1

},

{

"key": "蓝色",

"doc\_count": 1

}

]

}

}

}

## global bucket：单个品牌与所有品牌销量对比

aggregation，scope，一个聚合操作，必须在query的搜索结果范围内执行

出来两个结果，一个结果，是基于query搜索结果来聚合的; 一个结果，是对所有数据执行聚合的

GET /tvs/sales/\_search

{

"size": 0,

"query": {

"term": {

"brand": {

"value": "长虹"

}

}

},

"aggs": {

"single\_brand\_avg\_price": {

"avg": {

"field": "price"

}

},

"all": {

"global": {},

"aggs": {

"all\_brand\_avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

global：就是global bucket，就是将所有数据纳入聚合的scope，而不管之前的query

{

"took": 4,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 0,

"hits": []

},

"aggregations": {

"all": {

"doc\_count": 8,

"all\_brand\_avg\_price": {

"value": 2650

}

},

"single\_brand\_avg\_price": {

"value": 1666.6666666666667

}

}

}

single\_brand\_avg\_price：就是针对query搜索结果，执行的，拿到的，就是长虹品牌的平均价格

all.all\_brand\_avg\_price：拿到所有品牌的平均价格

## 统计价格大于1200的电视平均价格

搜索+聚合

过滤+聚合

GET /tvs/sales/\_search

{

"size": 0,

"query": {

"constant\_score": {

"filter": {

"range": {

"price": {

"gte": 1200

}

}

}

}

},

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

}

{

"took": 41,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 7,

"max\_score": 0,

"hits": []

},

"aggregations": {

"avg\_price": {

"value": 2885.714285714286

}

}

}

## 统计牌品最近一个月的平均价格

GET /tvs/sales/\_search

{

"size": 0,

"query": {

"term": {

"brand": {

"value": "长虹"

}

}

},

"aggs": {

"recent\_150d": {

"filter": {

"range": {

"sold\_date": {

"gte": "now-150d"

}

}

},

"aggs": {

"recent\_150d\_avg\_price": {

"avg": {

"field": "price"

}

}

}

},

"recent\_140d": {

"filter": {

"range": {

"sold\_date": {

"gte": "now-140d"

}

}

},

"aggs": {

"recent\_140d\_avg\_price": {

"avg": {

"field": "price"

}

}

}

},

"recent\_130d": {

"filter": {

"range": {

"sold\_date": {

"gte": "now-130d"

}

}

},

"aggs": {

"recent\_130d\_avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

aggs.filter，针对的是聚合去做的

如果放query里面的filter，是全局的，会对所有的数据都有影响

但是，如果，比如说，你要统计，长虹电视，最近1个月的平均值; 最近3个月的平均值; 最近6个月的平均值

bucket filter：对不同的bucket下的aggs，进行filter

## 按每种颜色的平均销售额降序排序

之前的话，排序，是按照每个bucket的doc\_count降序来排的

但是假如说，我们现在统计出来每个颜色的电视的销售额，需要按照销售额降序排序？？？？

GET /tvs/sales/\_search

{

"size": 0,

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color"

},

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

{

"took": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 8,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_color": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "红色",

"doc\_count": 4,

"avg\_price": {

"value": 3250

}

},

{

"key": "绿色",

"doc\_count": 2,

"avg\_price": {

"value": 2100

}

},

{

"key": "蓝色",

"doc\_count": 2,

"avg\_price": {

"value": 2000

}

}

]

}

}

}

GET /tvs/sales/\_search

{

"size": 0,

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color",

"order": {

"avg\_price": "asc"

}

},

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

## 颜色+品牌下钻分析时按最深层metric进行排序

GET /tvs/sales/\_search

{

"size": 0,

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color"

},

"aggs": {

"group\_by\_brand": {

"terms": {

"field": "brand",

"order": {

"avg\_price": "desc"

}

},

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

}

}

{

"took": 4,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 8,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_color": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "红色",

"doc\_count": 4,

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "三星",

"doc\_count": 1,

"avg\_price": {

"value": 8000

}

},

{

"key": "长虹",

"doc\_count": 3,

"avg\_price": {

"value": 1666.6666666666667

}

}

]

}

},

{

"key": "绿色",

"doc\_count": 2,

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "小米",

"doc\_count": 1,

"avg\_price": {

"value": 3000

}

},

{

"key": "TCL",

"doc\_count": 1,

"avg\_price": {

"value": 1200

}

}

]

}

},

{

"key": "蓝色",

"doc\_count": 2,

"group\_by\_brand": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "小米",

"doc\_count": 1,

"avg\_price": {

"value": 2500

}

},

{

"key": "TCL",

"doc\_count": 1,

"avg\_price": {

"value": 1500

}

}

]

}

}

]

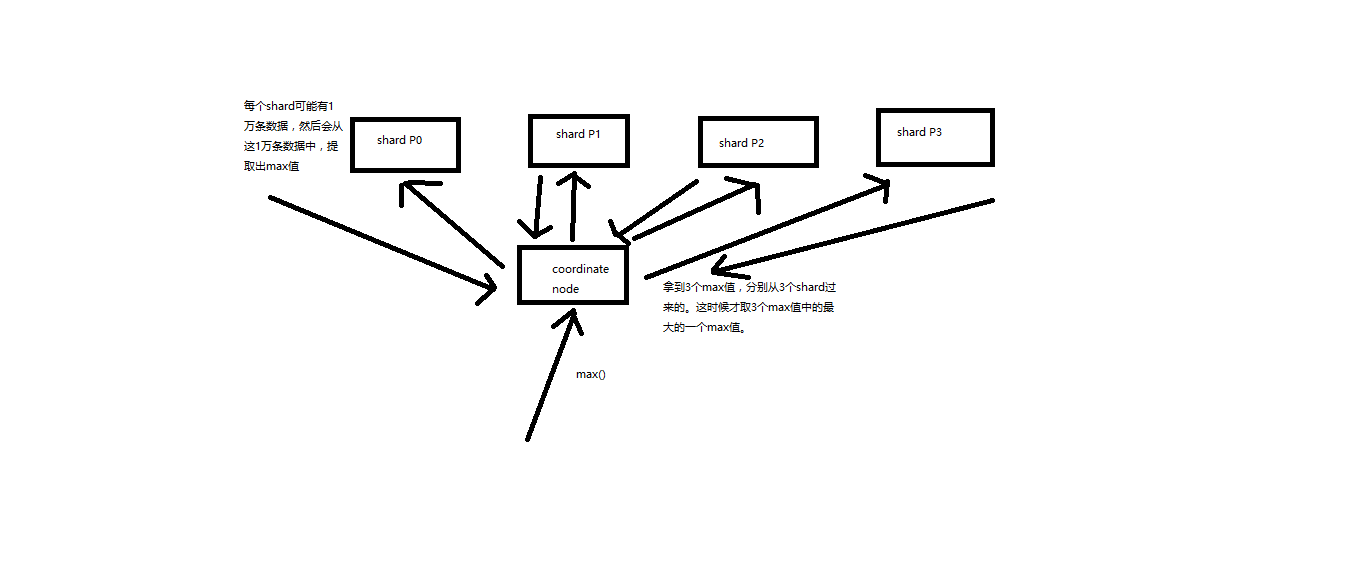
}

}

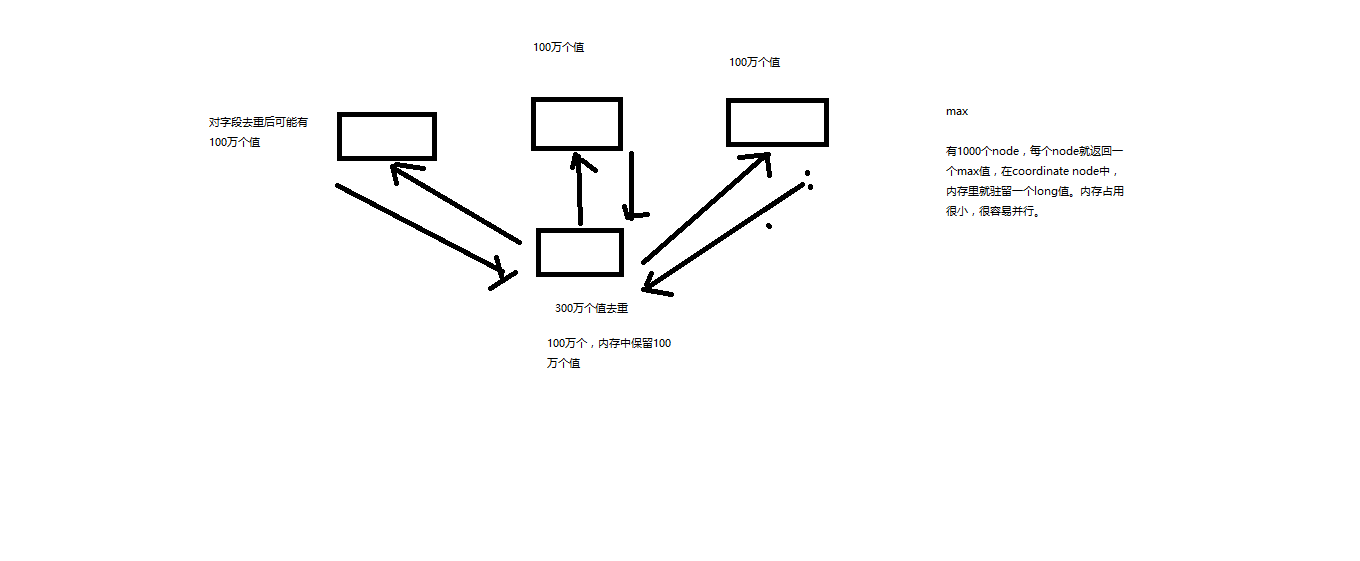
}

## 易并行聚合算法，三角选择原则，近似聚合算法

* 1. 画图讲解易并行聚合算法：max

**易并行算法.png**

有些聚合分析的算法，是很容易就可以并行的，比如说max

**不易并行的算法.png**

有些聚合分析的算法，是不好并行的，比如说，count(distinct)，并不是说，在每个node上，直接就出一些distinct value，就可以的，因为数据可能会很多

es会采取近似聚合的方式，就是采用在每个node上进行近估计的方式，得到最终的结论，cuont(distcint)，100万，1050万/95万 --> 5%左右的错误率

近似估计后的结果，不完全准确，但是速度会很快，一般会达到完全精准的算法的性能的数十倍

* 1. 三角选择原则

精准+实时+大数据 --> 选择2个

* 1. 精准+实时: 没有大数据，数据量很小，那么一般就是单击跑，随便你则么玩儿就可以
  2. 精准+大数据：hadoop，批处理，非实时，可以处理海量数据，保证精准，可能会跑几个小时
  3. 大数据+实时：es，不精准，近似估计，可能会有百分之几的错误率
  4. 近似聚合算法

如果采取近似估计的算法：延时在100ms左右，0.5%错误

如果采取100%精准的算法：延时一般在5s~几十s，甚至几十分钟，几小时， 0%错误

## cardinality去重算法及每月销售品牌数量统计

es，去重，cartinality metric，对每个bucket中的指定的field进行去重，取去重后的count，类似于count(distcint)

GET /tvs/sales/\_search

{

"size" : 0,

"aggs" : {

"months" : {

"date\_histogram": {

"field": "sold\_date",

"interval": "month"

},

"aggs": {

"distinct\_colors" : {

"cardinality" : {

"field" : "brand"

}

}

}

}

}

}

{

"took": 70,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 8,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_sold\_date": {

"buckets": [

{

"key\_as\_string": "2016-05-01T00:00:00.000Z",

"key": 1462060800000,

"doc\_count": 1,

"distinct\_brand\_cnt": {

"value": 1

}

},

{

"key\_as\_string": "2016-06-01T00:00:00.000Z",

"key": 1464739200000,

"doc\_count": 0,

"distinct\_brand\_cnt": {

"value": 0

}

},

{

"key\_as\_string": "2016-07-01T00:00:00.000Z",

"key": 1467331200000,

"doc\_count": 1,

"distinct\_brand\_cnt": {

"value": 1

}

},

{

"key\_as\_string": "2016-08-01T00:00:00.000Z",

"key": 1470009600000,

"doc\_count": 1,

"distinct\_brand\_cnt": {

"value": 1

}

},

{

"key\_as\_string": "2016-09-01T00:00:00.000Z",

"key": 1472688000000,

"doc\_count": 0,

"distinct\_brand\_cnt": {

"value": 0

}

},

{

"key\_as\_string": "2016-10-01T00:00:00.000Z",

"key": 1475280000000,

"doc\_count": 1,

"distinct\_brand\_cnt": {

"value": 1

}

},

{

"key\_as\_string": "2016-11-01T00:00:00.000Z",

"key": 1477958400000,

"doc\_count": 2,

"distinct\_brand\_cnt": {

"value": 1

}

},

{

"key\_as\_string": "2016-12-01T00:00:00.000Z",

"key": 1480550400000,

"doc\_count": 0,

"distinct\_brand\_cnt": {

"value": 0

}

},

{

"key\_as\_string": "2017-01-01T00:00:00.000Z",

"key": 1483228800000,

"doc\_count": 1,

"distinct\_brand\_cnt": {

"value": 1

}

},

{

"key\_as\_string": "2017-02-01T00:00:00.000Z",

"key": 1485907200000,

"doc\_count": 1,

"distinct\_brand\_cnt": {

"value": 1

}

}

]

}

}

}

## cardinality算法之优化内存开销以及HLL算法

cardinality，count(distinct)，5%的错误率，性能在100ms左右

* + - 1. precision\_threshold优化准确率和内存开销

GET /tvs/sales/\_search

{

"size" : 0,

"aggs" : {

"distinct\_brand" : {

"cardinality" : {

"field" : "brand",

"precision\_threshold" : 100

}

}

}

}

brand去重，如果brand的unique value，在100个以内，小米，长虹，三星，TCL，HTL。。。

在多少个unique value以内，cardinality，几乎保证100%准确

cardinality算法，会占用precision\_threshold \* 8 byte 内存消耗，100 \* 8 = 800个字节

占用内存很小。。。而且unique value如果的确在值以内，那么可以确保100%准确

100，数百万的unique value，错误率在5%以内

precision\_threshold，值设置的越大，占用内存越大，1000 \* 8 = 8000 / 1000 = 8KB，可以确保更多unique value的场景下，100%的准确

field，去重，count，这时候，unique value，10000，precision\_threshold=10000，10000 \* 8 = 80000个byte，80KB

2、HyperLogLog++ (HLL)算法性能优化

cardinality底层算法：HLL算法，HLL算法的性能

会对所有的uqniue value取hash值，通过hash值近似去求distcint count，误差

默认情况下，发送一个cardinality请求的时候，会动态地对所有的field value，取hash值; 将取hash值的操作，前移到建立索引的时候

PUT /tvs/

{

"mappings": {

"sales": {

"properties": {

"brand": {

"type": "text",

"fields": {

"hash": {

"type": "murmur3"

}

}

}

}

}

}

}

GET /tvs/sales/\_search

{

"size" : 0,

"aggs" : {

"distinct\_brand" : {

"cardinality" : {

"field" : "brand.hash",

"precision\_threshold" : 100

}

}

}

}

## percentiles百分比算法以及网站访问时延统计

需求：比如有一个网站，记录下了每次请求的访问的耗时，需要统计tp50，tp90，tp99

tp50：50%的请求的耗时最长在多长时间

tp90：90%的请求的耗时最长在多长时间

tp99：99%的请求的耗时最长在多长时间

PUT /websit e

{

"mappings": {

"logs": {

"properties": {

"latency": {

"type": "long"

},

"province": {

"type": "keyword"

},

"timestamp": {

"type": "date"

}

}

}

}

}

POST /website/logs/\_bulk

{ "index": {}}

{ "latency" : 105, "province" : "江苏", "timestamp" : "2016-10-28" }

{ "index": {}}

{ "latency" : 83, "province" : "江苏", "timestamp" : "2016-10-29" }

{ "index": {}}

{ "latency" : 92, "province" : "江苏", "timestamp" : "2016-10-29" }

{ "index": {}}

{ "latency" : 112, "province" : "江苏", "timestamp" : "2016-10-28" }

{ "index": {}}

{ "latency" : 68, "province" : "江苏", "timestamp" : "2016-10-28" }

{ "index": {}}

{ "latency" : 76, "province" : "江苏", "timestamp" : "2016-10-29" }

{ "index": {}}

{ "latency" : 101, "province" : "新疆", "timestamp" : "2016-10-28" }

{ "index": {}}

{ "latency" : 275, "province" : "新疆", "timestamp" : "2016-10-29" }

{ "index": {}}

{ "latency" : 166, "province" : "新疆", "timestamp" : "2016-10-29" }

{ "index": {}}

{ "latency" : 654, "province" : "新疆", "timestamp" : "2016-10-28" }

{ "index": {}}

{ "latency" : 389, "province" : "新疆", "timestamp" : "2016-10-28" }

{ "index": {}}

{ "latency" : 302, "province" : "新疆", "timestamp" : "2016-10-29" }

pencentiles

GET /website/logs/\_search

{

"size": 0,

"aggs": {

"latency\_percentiles": {

"percentiles": {

"field": "latency",

"percents": [

50,

95,

99

]

}

},

"latency\_avg": {

"avg": {

"field": "latency"

}

}

}

}

{

"took": 31,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 12,

"max\_score": 0,

"hits": []

},

"aggregations": {

"latency\_avg": {

"value": 201.91666666666666

},

"latency\_percentiles": {

"values": {

"50.0": 108.5,

"95.0": 508.24999999999983,

"99.0": 624.8500000000001

}

}

}

}

50%的请求，数值的最大的值是多少，不是完全准确的

GET /website/logs/\_search

{

"size": 0,

"aggs": {

"group\_by\_province": {

"terms": {

"field": "province"

},

"aggs": {

"latency\_percentiles": {

"percentiles": {

"field": "latency",

"percents": [

50,

95,

99

]

}

},

"latency\_avg": {

"avg": {

"field": "latency"

}

}

}

}

}

}

{

"took": 33,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 12,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_province": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "新疆",

"doc\_count": 6,

"latency\_avg": {

"value": 314.5

},

"latency\_percentiles": {

"values": {

"50.0": 288.5,

"95.0": 587.75,

"99.0": 640.75

}

}

},

{

"key": "江苏",

"doc\_count": 6,

"latency\_avg": {

"value": 89.33333333333333

},

"latency\_percentiles": {

"values": {

"50.0": 87.5,

"95.0": 110.25,

"99.0": 111.65

}

}

}

]

}

}

}

## percentiles rank及网站访问时延SLA统计

SLA：就是你提供的服务的标准

我们的网站的提供的访问延时的SLA，确保所有的请求100%，都必须在200ms以内，大公司内，一般都是要求100%在200ms以内

如果超过1s，则需要升级到A级故障，代表网站的访问性能和用户体验急剧下降

需求：在200ms以内的，有百分之多少，在1000毫秒以内的有百分之多少，percentile ranks metric

这个percentile ranks，其实比pencentile还要常用

按照品牌分组，计算，电视机，售价在1000占比，2000占比，3000占比

GET /website/logs/\_search

{

"size": 0,

"aggs": {

"group\_by\_province": {

"terms": {

"field": "province"

},

"aggs": {

"latency\_percentile\_ranks": {

"percentile\_ranks": {

"field": "latency",

"values": [

200,

1000

]

}

}

}

}

}

}

{

"took": 38,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 12,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_province": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "新疆",

"doc\_count": 6,

"latency\_percentile\_ranks": {

"values": {

"200.0": 29.40613026819923,

"1000.0": 100

}

}

},

{

"key": "江苏",

"doc\_count": 6,

"latency\_percentile\_ranks": {

"values": {

"200.0": 100,

"1000.0": 100

}

}

}

]

}

}

}

percentile的优化

TDigest算法，用很多节点来执行百分比的计算，近似估计，有误差，节点越多，越精准

compression

限制节点数量最多 compression \* 20 = 2000个node去计算

默认100

越大，占用内存越多，越精准，性能越差

一个节点占用32字节，100 \* 20 \* 32 = 64KB

如果你想要percentile算法越精准，compression可以设置的越大

## 基于doc value正排索引的聚合内部原理

聚合分析的内部原理是什么？？？？aggs，term，metric avg max，执行一个聚合操作的时候，内部原理是怎样的呢？用了什么样的数据结构去执行聚合？是不是用的倒排索引？

搜索+聚合，写个示例

GET /test\_index/test\_type/\_search

{

"query": {

"match": {

"search\_field": "test"

}

},

"aggs": {

"group\_by\_agg\_field": {

"terms": {

"field": "agg\_field"

}

}

}

}

纯用倒排索引来实现的弊端

es肯定不是纯用倒排索引来实现聚合+搜索的

search\_field

doc1: hello world test1, test2

doc2: hello test

doc3: world test

hello doc1,doc2

world doc1,doc3

test1 doc1

test2 doc1

test doc2,doc3

"query": {

"match": {

"search\_field": "test"

}

}

test --> doc2,doc3 --> search result, doc2,doc3

agg\_field

doc2: agg1

doc3: agg2

100万个值

...

...

...

...

agg1 doc2

agg2 doc3

doc2, doc3, search result --> 实际上，要搜索到doc2的agg\_field的值是多少，doc3的agg\_field的值是多少

doc2和doc3的agg\_field的值之后，就可以根据值进行分组，实现terms bucket操作

doc2的agg\_field的值是多少，这个时候，如果你手上只有一个倒排索引，你该怎么办？？？你要扫描整个倒排索引，去一个一个的搜，拿到每个值，比如说agg1，看一下，它是不是doc2的值，拿到agg2,看一下，是不是doc2的值，直到找到doc2的agg\_field的值，在倒排索引中

如果用纯倒排索引去实现聚合，现实不现实啊？？？性能是很低下的。。。搜索，search，搜倒排索引，搜那个term，就结束了。。。聚合，搜索出了1万个doc，每个doc都要在倒排索引中搜索出它的那个聚合field的值

倒排索引+正排索引（doc value）的原理和优势

doc value：正排索引

search\_field

doc1: hello world test1, test2

doc2: hello test

doc3: world test

hello doc1,doc2

world doc1,doc3

test1 doc1

test2 doc1

test doc2,doc3

"query": {

"match": {

"search\_field": "test"

}

}

test --> doc2,doc3 --> search result, doc2,doc3

doc value数据结构，正排索引

...

...

...

100万个

doc2: agg1

doc3: agg2

倒排索引的话，必须遍历完整个倒排索引才可以。。。。

因为可能你要聚合的那个field的值，是分词的，比如说hello world my name --> 一个doc的聚合field的值可能在倒排索引中对应多个value

所以说，当你在倒排索引中找到一个值，发现它是属于某个doc的时候，还不能停，必须遍历完整个倒排索引，才能说确保找到了每个doc对应的所有terms，然后进行分组聚合

...

...

...

100万个

doc2: agg1 hello world

doc3: agg2 test hello

我们有没有必要搜索完整个正排索引啊？？1万个doc --> 搜 -> 可能跟搜索到15000次，就搜索完了，就找到了1万个doc的聚合field的所有值了，然后就可以执行分组聚合操作了

## doc value机制内核级原理

* 1. doc value原理
  2. index-time生成

PUT/POST的时候，就会生成doc value数据，也就是正排索引

* 1. 核心原理与倒排索引类似

正排索引，也会写入磁盘文件中，然后呢，os cache先进行缓存，以提升

访问doc value正排索引的性能

如果os cache内存大小不足够放得下整个正排索引，doc value，就会

将doc value的数据写入磁盘文件中

* 1. 性能问题：给jvm更少内存，64g服务器，给jvm最多16g

es官方是建议，es大量是基于os cache来进行缓存和提升性能的，不

建议用jvm内存来进行缓存，那样会导致一定的gc开销和oom问题

给jvm更少的内存，给os cache更大的内存

64g服务器，给jvm最多16g，几十个g的内存给os cache

os cache可以提升doc value和倒排索引的缓存和查询效率

* 1. column压缩

doc1: 550

doc2: 550

doc3: 500

合并相同值，550，doc1和doc2都保留一个550的标识即可

* 1. 所有值相同，直接保留单值
  2. 少于256个值，使用table encoding模式：一种压缩方式
  3. 大于256个值，看有没有最大公约数，有就除以最大公约数，然后保留这个最大公约数

doc1: 36

doc2: 24

6 --> doc1: 6, doc2: 4 --> 保留一个最大公约数6的标识，6也保存起来

* 1. 如果没有最大公约数，采取offset结合压缩的方式：
  2. disable doc value

如果的确不需要doc value，比如聚合等操作，那么可以禁用，减少磁盘空间占

用

PUT my\_index

{

"mappings": {

"my\_type": {

"properties": {

"my\_field": {

"type": "keyword"

"doc\_values": false

}

}

}

}

}

## string field聚合实验以及fielddata原理

* 1. 对于分词的field执行aggregation，发现报错。。。

GET /test\_index/test\_type/\_search

{

"aggs": {

"group\_by\_test\_field": {

"terms": {

"field": "test\_field"

}

}

}

}

{

"error": {

"root\_cause": [

{

"type": "illegal\_argument\_exception",

"reason": "Fielddata is disabled on text fields by default. Set fielddata=true on [test\_field] in order to load fielddata in memory by uninverting the inverted index. Note that this can however use significant memory."

}

],

"type": "search\_phase\_execution\_exception",

"reason": "all shards failed",

"phase": "query",

"grouped": true,

"failed\_shards": [

{

"shard": 0,

"index": "test\_index",

"node": "4onsTYVZTjGvIj9\_spWz2w",

"reason": {

"type": "illegal\_argument\_exception",

"reason": "Fielddata is disabled on text fields by default. Set fielddata=true on [test\_field] in order to load fielddata in memory by uninverting the inverted index. Note that this can however use significant memory."

}

}

],

"caused\_by": {

"type": "illegal\_argument\_exception",

"reason": "Fielddata is disabled on text fields by default. Set fielddata=true on [test\_field] in order to load fielddata in memory by uninverting the inverted index. Note that this can however use significant memory."

}

},

"status": 400

}

对分词的field，直接执行聚合操作，会报错，大概意思是说，你必须要打开fielddata，然后将正排索引数据加载到内存中，才可以对分词的field执行聚合操作，而且会消耗很大的内存

* 1. 给分词的field，设置fielddata=true，发现可以执行，但是结果却。。。

POST /test\_index/\_mapping/test\_type

{

"properties": {

"test\_field": {

"type": "text",

"fielddata": true

}

}

}

{

"test\_index": {

"mappings": {

"test\_type": {

"properties": {

"test\_field": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

},

"fielddata": true

}

}

}

}

}

}

GET /test\_index/test\_type/\_search

{

"size": 0,

"aggs": {

"group\_by\_test\_field": {

"terms": {

"field": "test\_field"

}

}

}

}

{

"took": 23,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 2,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_test\_field": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "test",

"doc\_count": 2

}

]

}

}

}

如果要对分词的field执行聚合操作，必须将fielddata设置为true

* 1. 使用内置field不分词，对string field进行聚合

GET /test\_index/test\_type/\_search

{

"size": 0,

"aggs": {

"group\_by\_test\_field": {

"terms": {

"field": "test\_field.keyword"

}

}

}

}

{

"took": 3,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 2,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_test\_field": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "test",

"doc\_count": 2

}

]

}

}

}

如果对不分词的field执行聚合操作，直接就可以执行，不需要设置fieldata=true

* 1. 分词field+fielddata的工作原理

doc value --> 不分词的所有field，可以执行聚合操作 --> 如果你的某个field不分词，那么在index-time，就会自动生成doc value --> 针对这些不分词的field执行聚合操作的时候，自动就会用doc value来执行

分词field，是没有doc value的。。。在index-time，如果某个field是分词的，那么是不会给它建立doc value正排索引的，因为分词后，占用的空间过于大，所以默认是不支持分词field进行聚合的

分词field默认没有doc value，所以直接对分词field执行聚合操作，是会报错的

对于分词field，必须打开和使用fielddata，完全存在于纯内存中。。。结构和doc value类似。。。如果是ngram或者是大量term，那么必将占用大量的内存。。。

如果一定要对分词的field执行聚合，那么必须将fielddata=true，然后es就会在执行聚合操作的时候，现场将field对应的数据，建立一份fielddata正排索引，fielddata正排索引的结构跟doc value是类似的，但是只会讲fielddata正排索引加载到内存中来，然后基于内存中的fielddata正排索引执行分词field的聚合操作

如果直接对分词field执行聚合，报错，才会让我们开启fielddata=true，告诉我们，会将fielddata uninverted index，正排索引，加载到内存，会耗费内存空间

为什么fielddata必须在内存？因为大家自己思考一下，分词的字符串，需要按照term进行聚合，需要执行更加复杂的算法和操作，如果基于磁盘和os cache，那么性能会很差

## fielddata内存控制以及circuit breaker断路器

* 1. fielddata核心原理

fielddata加载到内存的过程是lazy加载的，对一个analzyed field执行聚合时，才会加载，而且是field-level加载的

一个index的一个field，所有doc都会被加载，而不是少数doc

不是index-time创建，是query-time创建

* 1. fielddata内存限制

indices.fielddata.cache.size: 20%，超出限制，清除内存已有fielddata数据

fielddata占用的内存超出了这个比例的限制，那么就清除掉内存中已有的fielddata数据

默认无限制，限制内存使用，但是会导致频繁evict和reload，大量IO性能损耗，以及内存碎片和gc

* 1. 监控fielddata内存使用

GET /\_stats/fielddata?fields=\*

GET /\_nodes/stats/indices/fielddata?fields=\*

GET /\_nodes/stats/indices/fielddata?level=indices&fields=\*

* 1. circuit breaker

如果一次query load的feilddata超过总内存，就会oom --> 内存溢出

circuit breaker会估算query要加载的fielddata大小，如果超出总内存，就短路，query直接失败

indices.breaker.fielddata.limit：fielddata的内存限制，默认60%

indices.breaker.request.limit：执行聚合的内存限制，默认40%

indices.breaker.total.limit：综合上面两个，限制在70%以内

## fielddata filter的细粒度内存加载控制

POST /test\_index/\_mapping/my\_type

{

"properties": {

"my\_field": {

"type": "text",

"fielddata": {

"filter": {

"frequency": {

"min": 0.01,

"min\_segment\_size": 500

}

}

}

}

}

}

min：仅仅加载至少在1%的doc中出现过的term对应的fielddata

比如说某个值，hello，总共有1000个doc，hello必须在10个doc中出现，那么这个hello对应的fielddata才会加载到内存中来

min\_segment\_size：少于500 doc的segment不加载fielddata

加载fielddata的时候，也是按照segment去进行加载的，某个segment里面的doc数量少于500个，那么这个segment的fielddata就不加载

这个，就我的经验来看，有点底层了，一般不会去设置它，大家知道就好

## fielddata预加载机制以及序号标记预加载

如果真的要对分词的field执行聚合，那么每次都在query-time现场生产fielddata并加载到内存中来，速度可能会比较慢

我们是不是可以预先生成加载fielddata到内存中来？？？

* 1. fielddata预加载

POST /test\_index/\_mapping/test\_type

{

"properties": {

"test\_field": {

"type": "string",

"fielddata": {

"loading" : "eager"

}

}

}

}

query-time的fielddata生成和加载到内存，变为index-time，建立倒排索引的时候，会同步生成fielddata并且加载到内存中来，这样的话，对分词field的聚合性能当然会大幅度增强

fielddata.loading: eager 可以告诉 Elasticsearch 预先将此

字段的内容载入内存中。

* 1. 序号标记预加载

global ordinal原理解释

doc1: status1

doc2: status2

doc3: status2

doc4: status1

有很多重复值的情况，会进行global ordinal标记

status1 --> 0

status2 --> 1

doc1: 0

doc2: 1

doc3: 1

doc4: 0

建立的fielddata也会是这个样子的，这样的好处就是减少重复字符串的出现的次数，减少内存的消耗

POST /test\_index/\_mapping/test\_type

{

"properties": {

"test\_field": {

"type": "string",

"fielddata": {

"loading" : "eager\_global\_ordinals"

}

}

}

}

## 海量bucket优化机制：从深度优先到广度优先

当buckets数量特别多的时候，深度优先和广度优先的原理，图解

我们的数据，是每个演员的每个电影的评论

每个演员的评论的数量 --> 每个演员的每个电影的评论的数量

评论数量排名前10个的演员 --> 每个演员的电影取到评论数量排名前5的电影

{

"aggs" : {

"actors" : {

"terms" : {

"field" : "actors",

"size" : 10,

"collect\_mode" : "breadth\_first"

},

"aggs" : {

"costars" : {

"terms" : {

"field" : "films",

"size" : 5

}

}

}

}

}

}

深度优先的方式去执行聚合操作的

actor1 actor2 .... actor

film1 film2 film3 film1 film2 film3 ...film

比如说，我们有10万个actor，最后其实是主要10个actor就可以了

但是我们已经深度优先的方式，构建了一整颗完整的树出来了，10万个actor，每个actor平均有10部电影，10万 + 100万 --> 110万的数据量的一颗树

裁剪掉10万个actor中的99990 actor，99990 \* 10 = film，剩下10个actor，每个actor的10个film裁剪掉5个，110万 --> 10 \* 5 = 50个

构建了大量的数据，然后裁剪掉了99.99%的数据，浪费了

广度优先的方式去执行聚合

actor1 actor2 actor3 ..... n个actor

10万个actor，不去构建它下面的film数据，10万 --> 99990，10个actor，构建出film，裁剪出其中的5个film即可，10万 -> 50个

10倍

## 关系型与document类型数据模型对比

关系型数据库的数据模型

es的document数据模型

public class Department {

private Integer deptId;

private String name;

private String desc;

private List<Employee> employees;

}

public class Employee {

private Integer empId;

private String name;

private Integer age;

private String gender;

private Department dept;

}

关系型数据库中

department表

dept\_id

name

desc

employee表

emp\_id

name

age

gender

dept\_id

三范式 --> 将每个数据实体拆分为一个独立的数据表，同时使用主外键关联关系将多个数据表关联起来 --> 确保没有任何冗余的数据

一份数据，只会放在一个数据表中 --> dept name，部门名称，就只会放在department表中，不会在employee表中也放一个dept name，如果说你要查看某个员工的部门名称，那么必须通过员工表中的外键，dept\_id，找到在部门表中对应的记录，然后找到部门名称

es文档数据模型

{

"deptId": "1",

"name": "研发部门",

"desc": "负责公司的所有研发项目",

"employees": [

{

"empId": "1",

"name": "张三",

"age": 28,

"gender": "男"

},

{

"empId": "2",

"name": "王兰",

"age": 25,

"gender": "女"

},

{

"empId": "3",

"name": "李四",

"age": 34,

"gender": "男"

}

]

}

es，更加类似于面向对象的数据模型，将所有由关联关系的数据，放在一个doc json类型数据中，整个数据的关系，还有完整的数据，都放在了一起

## 通过应用层join实现用户与博客的关联

* 1. 构造用户与博客数据

在构造数据模型的时候，还是将有关联关系的数据，然后分割为不同的实体，类似于关系型数据库中的模型

案例背景：博客网站， 我们会模拟各种用户发表各种博客，然后针对用户和博客之间的关系进行数据建模，同时针对建模好的数据执行各种搜索/聚合的操作

PUT /website/users/1

{

"name": "小鱼儿",

"email": "xiaoyuer@sina.com",

"birthday": "1980-01-01"

}

PUT /website/blogs/1

{

"title": "我的第一篇博客",

"content": "这是我的第一篇博客，开通啦！！！"

"userId": 1

}

一个用户对应多个博客，一对多的关系，做了建模

建模方式，分割实体，类似三范式的方式，用主外键关联关系，将多个实体关联起来

* 1. 搜索小鱼儿发表的所有博客

GET /website/users/\_search

{

"query": {

"term": {

"name.keyword": {

"value": "小鱼儿"

}

}

}

}

{

"took": 91,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 0.2876821,

"hits": [

{

"\_index": "website",

"\_type": "users",

"\_id": "1",

"\_score": 0.2876821,

"\_source": {

"name": "小鱼儿",

"email": "xiaoyuer@sina.com",

"birthday": "1980-01-01"

}

}

]

}

}

比如这里搜索的是，1万个用户的博客，可能第一次搜索，会得到1万个userId

GET /website/blogs/\_search

{

"query": {

"constant\_score": {

"filter": {

"terms": {

"userId": [

1

]

}

}

}

}

}

第二次搜索的时候，要放入terms中1万个userId，才能进行搜索，这个时候性能比较差了

{

"took": 4,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1,

"hits": [

{

"\_index": "website",

"\_type": "blogs",

"\_id": "1",

"\_score": 1,

"\_source": {

"title": "小鱼儿的第一篇博客",

"content": "大家好，我是小鱼儿，这是我写的第一篇博客！",

"userId": 1

}

}

]

}

}

上面的操作，就属于应用层的join，在应用层先查出一份数据，然后再查出一份数据，进行关联

* 1. 优点和缺点

优点：数据不冗余，维护方便

缺点：应用层join，如果关联数据过多，导致查询过大，性能很差

## 通过数据冗余实现用户与博客的关联

* + - 1. 构造冗余的用户和博客数据

第二种建模方式：用冗余数据，采用文档数据模型，进行数据建模，实现用户和博客的关联

PUT /website/users/1

{

"name": "小鱼儿",

"email": "xiaoyuer@sina.com",

"birthday": "1980-01-01"

}

PUT /website/blogs/1

{

"title": "小鱼儿的第一篇博客",

"content": "大家好，我是小鱼儿。。。",

"userInfo": {

"userId": 1,

"username": "小鱼儿"

}

}

冗余数据，就是说，将可能会进行搜索的条件和要搜索的数据，放在一个doc

中

* + 1. 基于冗余用户数据搜索博客

GET /website/blogs/\_search

{

"query": {

"term": {

"userInfo.username.keyword": {

"value": "小鱼儿"

}

}

}

}

就不需要走应用层的join，先搜一个数据，找到id，再去搜另一份数据

直接走一个有冗余数据的type即可，指定要的搜索条件，即可搜索出自己想要的数据来

* + 1. 优点和缺点

优点：性能高，不需要执行两次搜索

缺点：数据冗余，维护成本高 --> 每次如果你的username变化了，同时要更新user type和blog type

一般来说，对于es这种NoSQL类型的数据存储来讲，都是冗余模式....

当然，你要去维护数据的关联关系，也是很有必要的，所以一旦出现冗余数据的修改，必须记得将所有关联的数据全部更新

## 对每个用户发表的博客进行分组

1. 构造更多测试数据

PUT /website/users/3

{

"name": "黄药师",

"email": "huangyaoshi@sina.com",

"birthday": "1970-10-24"

}

PUT /website/blogs/3

{

"title": "我是黄药师",

"content": "我是黄药师啊，各位同学们！！！",

"userInfo": {

"userId": 1,

"userName": "黄药师"

}

}

PUT /website/users/2

{

"name": "花无缺",

"email": "huawuque@sina.com",

"birthday": "1980-02-02"

}

PUT /website/blogs/4

{

"title": "花无缺的身世揭秘",

"content": "大家好，我是花无缺，所以我的身世是。。。",

"userInfo": {

"userId": 2,

"userName": "花无缺"

}

}

1. 对每个用户发表的博客进行分组

比如说，小鱼儿发表的那些博客，花无缺发表了哪些博客，黄药师发表了哪些博客

GET /website/blogs/\_search

{

"size": 0,

"aggs": {

"group\_by\_username": {

"terms": {

"field": "userInfo.username.keyword"

},

"aggs": {

"top\_blogs": {

"top\_hits": {

"\_source": {

"include": "title"

},

"size": 5

}

}

}

}

}

}

## 对文件系统进行数据建模以及文件搜索实战

数据建模，对类似文件系统这种的有多层级关系的数据进行建模

1. 文件系统数据构造

PUT /fs

{

"settings": {

"analysis": {

"analyzer": {

"paths": {

"tokenizer": "path\_hierarchy"

}

}

}

}

}

path\_hierarchy tokenizer讲解

path\_hierarchy tokenizer：路径分词器

/a/b/c/d --> path\_hierarchy -> /a/b/c/d, /a/b/c, /a/b, /a

fs: filesystem

PUT /fs/\_mapping/file

{

"properties": {

"name": {

"type": "keyword"

},

"path": {

"type": "keyword",

"fields": {

"tree": {

"type": "text",

"analyzer": "paths"

}

}

}

}

}

PUT /fs/file/1

{

"name": "README.txt",

"path": "/workspace/projects/helloworld",

"contents": "这是我的第一个elasticsearch程序"

}

1. 对文件系统执行搜索

文件搜索需求：查找一份，内容包括elasticsearch，在/workspace/projects/hellworld这个目录下的文件

GET /fs/file/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"contents": "elasticsearch"

}

},

{

"constant\_score": {

"filter": {

"term": {

"path": "/workspace/projects/helloworld"

}

}

}

}

]

}

}

}

{

"took": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1.284885,

"hits": [

{

"\_index": "fs",

"\_type": "file",

"\_id": "1",

"\_score": 1.284885,

"\_source": {

"name": "README.txt",

"path": "/workspace/projects/helloworld",

"contents": "这是我的第一个elasticsearch程序"

}

}

]

}

}

搜索需求2：搜索/workspace目录下，内容包含elasticsearch的所有的文件

/workspace/projects/helloworld doc1

/workspace/projects doc1

/workspace doc1

GET /fs/file/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"contents": "elasticsearch"

}

},

{

"constant\_score": {

"filter": {

"term": {

"path.tree": "/workspace"

}

}

}

}

]

}

}

}

## 基于全局锁实现悲观锁并发控制

1. 悲观锁的简要说明

基于version的乐观锁并发控制

在数据建模，结合文件系统建模的这个案例，把悲观锁的并发控制，3种锁粒度，都给大家仔细讲解一下

最粗的一个粒度，全局锁

/workspace/projects/helloworld

如果多个线程，都过来，要并发地给/workspace/projects/helloworld下的README.txt修改文件名

实际上要进行并发的控制，避免出现多线程的并发安全问题，比如多个线程修改，纯并发，先执行的修改操作被后执行的修改操作给覆盖了

get current version

带着这个current version去执行修改，如果一旦发现数据已经被别人给修改了，version号跟之前自己获取的已经不一样了; 那么必须重新获取新的version号再次尝试修改

上来就尝试给这条数据加个锁，然后呢，此时就只有你能执行各种各样的操作了，其他人不能执行操作

第一种锁：全局锁，直接锁掉整个fs index

1. 全局锁的上锁实验

PUT /fs/lock/global/\_create

{}

fs: 你要上锁的那个index

lock: 就是你指定的一个对这个index上全局锁的一个type

global: 就是你上的全局锁对应的这个doc的id

\_create：强制必须是创建，如果/fs/lock/global这个doc已经存在，那么创建失败，报错

利用了doc来进行上锁

/fs/lock/global /index/type/id --> doc

{

"\_index": "fs",

"\_type": "lock",

"\_id": "global",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

另外一个线程同时尝试上锁

PUT /fs/lock/global/\_create

{}

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[lock][global]: version conflict, document already exists (current version [1])",

"index\_uuid": "IYbj0OLGQHmMUpLfbhD4Hw",

"shard": "2",

"index": "fs"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[lock][global]: version conflict, document already exists (current version [1])",

"index\_uuid": "IYbj0OLGQHmMUpLfbhD4Hw",

"shard": "2",

"index": "fs"

},

"status": 409

}

如果失败，就再次重复尝试上锁

执行各种操作。。。

POST /fs/file/1/\_update

{

"doc": {

"name": "README1.txt"

}

}

{

"\_index": "fs",

"\_type": "file",

"\_id": "1",

"\_version": 2,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

DELETE /fs/lock/global

{

"found": true,

"\_index": "fs",

"\_type": "lock",

"\_id": "global",

"\_version": 2,

"result": "deleted",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

另外一个线程，因为之前发现上锁失败，反复尝试重新上锁，终于上锁成功了，因为之前获取到全局锁的那个线程已经delete /fs/lock/global全局锁了

PUT /fs/lock/global/\_create

{}

{

"\_index": "fs",

"\_type": "lock",

"\_id": "global",

"\_version": 3,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

POST /fs/file/1/\_update

{

"doc": {

"name": "README.txt"

}

}

{

"\_index": "fs",

"\_type": "file",

"\_id": "1",

"\_version": 3,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

DELETE /fs/lock/global

1. 全局锁的优点和缺点

优点：操作非常简单，非常容易使用，成本低

缺点：你直接就把整个index给上锁了，这个时候对index中所有的doc的操作，都会被block住，导致整个系统的并发能力很低

上锁解锁的操作不是频繁，然后每次上锁之后，执行的操作的耗时不会太长，用这种方式，方便

## 基于document锁实现悲观锁并发控制

1. 对document level锁，详细的讲解

全局锁，一次性就锁整个index，对这个index的所有增删改操作都会被block住，如果上锁不频繁，还可以，比较简单

细粒度的一个锁，document锁，顾名思义，每次就锁你要操作的，你要执行增删改的那些doc，doc锁了，其他线程就不能对这些doc执行增删改操作了

但是你只是锁了部分doc，其他线程对其他的doc还是可以上锁和执行增删改操作的

document锁，是用脚本进行上锁

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 123 },

"script": "if ( ctx.\_source.process\_id != process\_id ) { assert false }; ctx.op = 'noop';"

"params": {

"process\_id": 123

}

}

/fs/lock，是固定的，就是说fs下的lock type，专门用于进行上锁

/fs/lock/id，比如1，id其实就是你要上锁的那个doc的id，代表了某个doc数据对应的lock（也是一个doc）

\_update + upsert：执行upsert操作

params，里面有个process\_id，process\_id，是你的要执行增删改操作的进程的唯一id，比如说可以在java系统，启动的时候，给你的每个线程都用UUID自动生成一个thread id，你的系统进程启动的时候给整个进程也分配一个UUID。process\_id + thread\_id就代表了某一个进程下的某个线程的唯一标识。可以自己用UUID生成一个唯一ID

process\_id很重要，会在lock中，设置对对应的doc加锁的进程的id，这样其他进程过来的时候，才知道，这条数据已经被别人给锁了

assert false，不是当前进程加锁的话，则抛出异常

ctx.op='noop'，不做任何修改

如果该document之前没有被锁，/fs/lock/1之前不存在，也就是doc id=1没有被别人上过锁; upsert的语法，那么执行index操作，创建一个/fs/lock/id这条数据，而且用params中的数据作为这个lock的数据。process\_id被设置为123，script不执行。这个时候象征着process\_id=123的进程已经锁了一个doc了。

如果document被锁了，就是说/fs/lock/1已经存在了，代表doc id=1已经被某个进程给锁了。那么执行update操作，script，此时会比对process\_id，如果相同，就是说，某个进程，之前锁了这个doc，然后这次又过来，就可以直接对这个doc执行操作，说明是该进程之前锁的doc，则不报错，不执行任何操作，返回success; 如果process\_id比对不上，说明doc被其他doc给锁了，此时报错

/fs/lock/1

{

"process\_id": 123

}

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 123 },

"script": "if ( ctx.\_source.process\_id != process\_id ) { assert false }; ctx.op = 'noop';"

"params": {

"process\_id": 123

}

}

script：ctx.\_source.process\_id，123

process\_id：加锁的upsert请求中带过来额proess\_id

如果两个process\_id相同，说明是一个进程先加锁，然后又过来尝试加锁，可能是要执行另外一个操作，此时就不会block，对同一个process\_id是不会block，ctx.op= 'noop'，什么都不做，返回一个success

如果说已经有一个进程加了锁了

/fs/lock/1

{

"process\_id": 123

}

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 123 },

"script": "if ( ctx.\_source.process\_id != process\_id ) { assert false }; ctx.op = 'noop';"

"params": {

"process\_id": 234

}

}

"script": "if ( ctx.\_source.process\_id != process\_id ) { assert false }; ctx.op = 'noop';"

ctx.\_source.process\_id：123

process\_id: 234

process\_id不相等，说明这个doc之前已经被别人上锁了，process\_id=123上锁了; process\_id=234过来再次尝试上锁，失败，assert false，就会报错

此时遇到报错的process，就应该尝试重新上锁，直到上锁成功

有报错的话，如果有些doc被锁了，那么需要重试

直到所有锁定都成功，执行自己的操作。。。

释放所有的锁

1. 上document锁的完整实验过程

scripts/judge-lock.groovy: if ( ctx.\_source.process\_id != process\_id ) { assert false }; ctx.op = 'noop';

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 123 },

"script": {

"lang": "groovy",

"file": "judge-lock",

"params": {

"process\_id": 123

}

}

}

{

"\_index": "fs",

"\_type": "lock",

"\_id": "1",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

GET /fs/lock/1

{

"\_index": "fs",

"\_type": "lock",

"\_id": "1",

"\_version": 1,

"found": true,

"\_source": {

"process\_id": 123

}

}

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 234 },

"script": {

"lang": "groovy",

"file": "judge-lock",

"params": {

"process\_id": 234

}

}

}

{

"error": {

"root\_cause": [

{

"type": "remote\_transport\_exception",

"reason": "[4onsTYV][127.0.0.1:9300][indices:data/write/update[s]]"

}

],

"type": "illegal\_argument\_exception",

"reason": "failed to execute script",

"caused\_by": {

"type": "script\_exception",

"reason": "error evaluating judge-lock",

"caused\_by": {

"type": "power\_assertion\_error",

"reason": "assert false\n"

},

"script\_stack": [],

"script": "",

"lang": "groovy"

}

},

"status": 400

}

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 123 },

"script": {

"lang": "groovy",

"file": "judge-lock",

"params": {

"process\_id": 123

}

}

}

{

"\_index": "fs",

"\_type": "lock",

"\_id": "1",

"\_version": 1,

"result": "noop",

"\_shards": {

"total": 0,

"successful": 0,

"failed": 0

}

}

POST /fs/file/1/\_update

{

"doc": {

"name": "README1.txt"

}

}

{

"\_index": "fs",

"\_type": "file",

"\_id": "1",

"\_version": 4,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

POST /fs/\_refresh

GET /fs/lock/\_search?scroll=1m

{

"query": {

"term": {

"process\_id": 123

}

}

}

{

"\_scroll\_id": "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAACPkFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAj5RY0b25zVFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAAI-YWNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAAACPnFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAj6BY0b25zVFlWWlRqR3ZJajlfc3BXejJ3",

"took": 51,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1,

"hits": [

{

"\_index": "fs",

"\_type": "lock",

"\_id": "1",

"\_score": 1,

"\_source": {

"process\_id": 123

}

}

]

}

}

PUT /fs/lock/\_bulk

{ "delete": { "\_id": 1}}

{

"took": 20,

"errors": false,

"items": [

{

"delete": {

"found": true,

"\_index": "fs",

"\_type": "lock",

"\_id": "1",

"\_version": 2,

"result": "deleted",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"status": 200

}

}

]

}

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 234 },

"script": {

"lang": "groovy",

"file": "judge-lock",

"params": {

"process\_id": 234

}

}

}

process\_id=234上锁就成功了

## 基于共享锁和排他锁实现悲观锁并发控制

* 1. 共享锁和排他锁的说明

共享锁：这份数据是共享的，然后多个线程过来，都可以获取同一个数据的共享锁，然后对这个数据执行读操作

排他锁：是排他的操作，只能一个线程获取排他锁，然后执行增删改操作

读写锁的分离

如果只是要读取数据的话，那么任意个线程都可以同时进来然后读取数据，每个线程都可以上一个共享锁

但是这个时候，如果有线程要过来修改数据，那么会尝试上排他锁，排他锁会跟共享锁互斥，也就是说，如果有人已经上了共享锁了，那么排他锁就不能上，就得等

如果有人在读数据，就不允许别人来修改数据

反之，也是一样的

如果有人在修改数据，就是加了排他锁

那么其他线程过来要修改数据，也会尝试加排他锁，此时会失败，锁冲突，必须等待，同时只能有一个线程修改数据

如果有人过来同时要读取数据，那么会尝试加共享锁，此时会失败，因为共享锁和排他锁是冲突的

如果有在修改数据，就不允许别人来修改数据，也不允许别人来读取数据

* 1. 共享锁和排他锁的实验

第一步：有人在读数据，其他人也能过来读数据

judge-lock-2.groovy: if (ctx.\_source.lock\_type == 'exclusive') { assert false }; ctx.\_source.lock\_count++

POST /fs/lock/1/\_update

{

"upsert": {

"lock\_type": "shared",

"lock\_count": 1

},

"script": {

"lang": "groovy",

"file": "judge-lock-2"

}

}

POST /fs/lock/1/\_update

{

"upsert": {

"lock\_type": "shared",

"lock\_count": 1

},

"script": {

"lang": "groovy",

"file": "judge-lock-2"

}

}

GET /fs/lock/1

{

"\_index": "fs",

"\_type": "lock",

"\_id": "1",

"\_version": 3,

"found": true,

"\_source": {

"lock\_type": "shared",

"lock\_count": 3

}

}

就给大家模拟了，有人上了共享锁，你还是要上共享锁，直接上就行了，没问题，只是lock\_count加1

* 1. 已经有人上了共享锁，然后有人要上排他锁

PUT /fs/lock/1/\_create

{ "lock\_type": "exclusive" }

排他锁用的不是upsert语法，create语法，要求lock必须不能存在，直接自己是第一个上锁的人，上的是排他锁

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[lock][1]: version conflict, document already exists (current version [3])",

"index\_uuid": "IYbj0OLGQHmMUpLfbhD4Hw",

"shard": "3",

"index": "fs"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[lock][1]: version conflict, document already exists (current version [3])",

"index\_uuid": "IYbj0OLGQHmMUpLfbhD4Hw",

"shard": "3",

"index": "fs"

},

"status": 409

}

如果已经有人上了共享锁，明显/fs/lock/1是存在的，create语法去上排他锁，肯定会报错

* 1. 对共享锁进行解锁

POST /fs/lock/1/\_update

{

"script": {

"lang": "groovy",

"file": "unlock-shared"

}

}

连续解锁3次，此时共享锁就彻底没了

每次解锁一个共享锁，就对lock\_count先减1，如果减了1之后，是0，那么说明所有的共享锁都解锁完了，此时就就将/fs/lock/1删除，就彻底解锁所有的共享锁

3、上排他锁，再上排他锁

PUT /fs/lock/1/\_create

{ "lock\_type": "exclusive" }

其他线程

PUT /fs/lock/1/\_create

{ "lock\_type": "exclusive" }

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[lock][1]: version conflict, document already exists (current version [7])",

"index\_uuid": "IYbj0OLGQHmMUpLfbhD4Hw",

"shard": "3",

"index": "fs"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[lock][1]: version conflict, document already exists (current version [7])",

"index\_uuid": "IYbj0OLGQHmMUpLfbhD4Hw",

"shard": "3",

"index": "fs"

},

"status": 409

}

* 1. 上排他锁，上共享锁

POST /fs/lock/1/\_update

{

"upsert": {

"lock\_type": "shared",

"lock\_count": 1

},

"script": {

"lang": "groovy",

"file": "judge-lock-2"

}

}

{

"error": {

"root\_cause": [

{

"type": "remote\_transport\_exception",

"reason": "[4onsTYV][127.0.0.1:9300][indices:data/write/update[s]]"

}

],

"type": "illegal\_argument\_exception",

"reason": "failed to execute script",

"caused\_by": {

"type": "script\_exception",

"reason": "error evaluating judge-lock-2",

"caused\_by": {

"type": "power\_assertion\_error",

"reason": "assert false\n"

},

"script\_stack": [],

"script": "",

"lang": "groovy"

}

},

"status": 400

}

* 1. 解锁排他锁

DELETE /fs/lock/1

## 基于nested object实现博客与评论嵌套关系

* 1. 做一个实验，引出来为什么需要nested object

冗余数据方式的来建模，其实用的就是object类型，我们这里又要引入一种新的object类型，nested object类型

博客，评论，做的这种数据模型

PUT /website/blogs/6

{

"title": "花无缺发表的一篇帖子",

"content": "我是花无缺，大家要不要考虑一下投资房产和买股票的事情啊。。。",

"tags": [ "投资", "理财" ],

"comments": [

{

"name": "小鱼儿",

"comment": "什么股票啊？推荐一下呗",

"age": 28,

"stars": 4,

"date": "2016-09-01"

},

{

"name": "黄药师",

"comment": "我喜欢投资房产，风，险大收益也大",

"age": 31,

"stars": 5,

"date": "2016-10-22"

}

]

}

被年龄是28岁的黄药师评论过的博客，搜索

GET /website/blogs/\_search

{

"query": {

"bool": {

"must": [

{ "match": { "comments.name": "黄药师" }},

{ "match": { "comments.age": 28 }}

]

}

}

}

{

"took": 102,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1.8022683,

"hits": [

{

"\_index": "website",

"\_type": "blogs",

"\_id": "6",

"\_score": 1.8022683,

"\_source": {

"title": "花无缺发表的一篇帖子",

"content": "我是花无缺，大家要不要考虑一下投资房产和买股票的事情啊。。。",

"tags": [

"投资",

"理财"

],

"comments": [

{

"name": "小鱼儿",

"comment": "什么股票啊？推荐一下呗",

"age": 28,

"stars": 4,

"date": "2016-09-01"

},

{

"name": "黄药师",

"comment": "我喜欢投资房产，风，险大收益也大",

"age": 31,

"stars": 5,

"date": "2016-10-22"

}

]

}

}

]

}

}

结果是。。。好像不太对啊？？？

object类型数据结构的底层存储。。。

{

"title": [ "花无缺", "发表", "一篇", "帖子" ],

"content": [ "我", "是", "花无缺", "大家", "要不要", "考虑", "一下", "投资", "房产", "买", "股票", "事情" ],

"tags": [ "投资", "理财" ],

"comments.name": [ "小鱼儿", "黄药师" ],

"comments.comment": [ "什么", "股票", "推荐", "我", "喜欢", "投资", "房产", "风险", "收益", "大" ],

"comments.age": [ 28, 31 ],

"comments.stars": [ 4, 5 ],

"comments.date": [ 2016-09-01, 2016-10-22 ]

}

object类型底层数据结构，会将一个json数组中的数据，进行扁平化

所以，直接命中了这个document，name=黄药师，age=28，正好符合

* 1. 引入nested object类型，来解决object类型底层数据结构导致的问题

修改mapping，将comments的类型从object设置为nested

PUT /website

{

"mappings": {

"blogs": {

"properties": {

"comments": {

"type": "nested",

"properties": {

"name": { "type": "string" },

"comment": { "type": "string" },

"age": { "type": "short" },

"stars": { "type": "short" },

"date": { "type": "date" }

}

}

}

}

}

}

{

"comments.name": [ "小鱼儿" ],

"comments.comment": [ "什么", "股票", "推荐" ],

"comments.age": [ 28 ],

"comments.stars": [ 4 ],

"comments.date": [ 2014-09-01 ]

}

{

"comments.name": [ "黄药师" ],

"comments.comment": [ "我", "喜欢", "投资", "房产", "风险", "收益", "大" ],

"comments.age": [ 31 ],

"comments.stars": [ 5 ],

"comments.date": [ 2014-10-22 ]

}

{

"title": [ "花无缺", "发表", "一篇", "帖子" ],

"body": [ "我", "是", "花无缺", "大家", "要不要", "考虑", "一下", "投资", "房产", "买", "股票", "事情" ],

"tags": [ "投资", "理财" ]

}

再次搜索，成功了。。。

GET /website/blogs/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"title": "花无缺"

}

},

{

"nested": {

"path": "comments",

"query": {

"bool": {

"must": [

{

"match": {

"comments.name": "黄药师"

}

},

{

"match": {

"comments.age": 28

}

}

]

}

}

}

}

]

}

}

}

score\_mode：max，min，avg，none，默认是avg

## 对嵌套的博客评论数据进行聚合分析

基于nested object中的数据进行聚合分析

聚合数据分析的需求1：按照评论日期进行bucket划分，然后拿到每个月的评论的评分的平均值

GET /website/blogs/\_search

{

"size": 0,

"aggs": {

"comments\_path": {

"nested": {

"path": "comments"

},

"aggs": {

"group\_by\_comments\_date": {

"date\_histogram": {

"field": "comments.date",

"interval": "month",

"format": "yyyy-MM"

},

"aggs": {

"avg\_stars": {

"avg": {

"field": "comments.stars"

}

}

}

}

}

}

}

}

{

"took": 52,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 2,

"max\_score": 0,

"hits": []

},

"aggregations": {

"comments\_path": {

"doc\_count": 4,

"group\_by\_comments\_date": {

"buckets": [

{

"key\_as\_string": "2016-08",

"key": 1470009600000,

"doc\_count": 1,

"avg\_stars": {

"value": 3

}

},

{

"key\_as\_string": "2016-09",

"key": 1472688000000,

"doc\_count": 2,

"avg\_stars": {

"value": 4.5

}

},

{

"key\_as\_string": "2016-10",

"key": 1475280000000,

"doc\_count": 1,

"avg\_stars": {

"value": 5

}

}

]

}

}

}

}

GET /website/blogs/\_search

{

"size": 0,

"aggs": {

"comments\_path": {

"nested": {

"path": "comments"

},

"aggs": {

"group\_by\_comments\_age": {

"histogram": {

"field": "comments.age",

"interval": 10

},

"aggs": {

"reverse\_path": {

"reverse\_nested": {},

"aggs": {

"group\_by\_tags": {

"terms": {

"field": "tags.keyword"

}

}

}

}

}

}

}

}

}

}

{

"took": 5,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 2,

"max\_score": 0,

"hits": []

},

"aggregations": {

"comments\_path": {

"doc\_count": 4,

"group\_by\_comments\_age": {

"buckets": [

{

"key": 20,

"doc\_count": 1,

"reverse\_path": {

"doc\_count": 1,

"group\_by\_tags": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "投资",

"doc\_count": 1

},

{

"key": "理财",

"doc\_count": 1

}

]

}

}

},

{

"key": 30,

"doc\_count": 3,

"reverse\_path": {

"doc\_count": 2,

"group\_by\_tags": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "大侠",

"doc\_count": 1

},

{

"key": "投资",

"doc\_count": 1

},

{

"key": "理财",

"doc\_count": 1

},

{

"key": "练功",

"doc\_count": 1

}

]

}

}

}

]

}

}

reverse\_nested：连接回根/主文档级别的聚合，因为没有path定义。如果已在映射中定义了多个分层嵌套对象类型，则通过该path选项，reverse\_nested聚合可以连接回不同的级别

## 研发中心管理案例以及父子关系数据建模

nested object的建模，有个不好的地方，就是采取的是类似冗余数据的方式，将多个数据都放在一起了，维护成本就比较高

parent child建模方式，采取的是类似于关系型数据库的三范式类的建模，多个实体都分割开来，每个实体之间都通过一些关联方式，进行了父子关系的关联，各种数据不需要都放在一起，父doc和子doc分别在进行更新的时候，都不会影响对方

一对多关系的建模，维护起来比较方便，而且我们之前说过，类似关系型数据库的建模方式，应用层join的方式，会导致性能比较差，因为做多次搜索。父子关系的数据模型，不会，性能很好。因为虽然数据实体之间分割开来，但是我们在搜索的时候，由es自动为我们处理底层的关联关系，并且通过一些手段保证搜索性能。

父子关系数据模型，相对于nested数据模型来说，优点是父doc和子doc互相之间不会影响

要点：父子关系元数据映射，用于确保查询时候的高性能，但是有一个限制，就是父子数据必须存在于一个shard中

父子关系数据存在一个shard中，而且还有映射其关联关系的元数据，那么搜索父子关系数据的时候，不用跨分片，一个分片本地自己就搞定了，性能当然高咯

案例背景：研发中心员工管理案例，一个IT公司有多个研发中心，每个研发中心有多个员工

PUT /company

{

"mappings": {

"rd\_center": {},

"employee": {

"\_parent": {

"type": "rd\_center"

}

}

}

}

父子关系建模的核心，多个type之间有父子关系，用\_parent指定父type

POST /company/rd\_center/\_bulk

{ "index": { "\_id": "1" }}

{ "name": "北京研发总部", "city": "北京", "country": "中国" }

{ "index": { "\_id": "2" }}

{ "name": "上海研发中心", "city": "上海", "country": "中国" }

{ "index": { "\_id": "3" }}

{ "name": "硅谷人工智能实验室", "city": "硅谷", "country": "美国" }

shard路由的时候，id=1的rd\_center doc，默认会根据id进行路由，到某一个shard

PUT /company/employee/1?parent=1

{

"name": "张三",

"birthday": "1970-10-24",

"hobby": "爬山"

}

维护父子关系的核心，parent=1，指定了这个数据的父doc的id

此时，parent-child关系，就确保了说，父doc和子doc都是保存在一个shard上的。内部原理还是doc routing，employee和rd\_center的数据，都会用parent id作为routing，这样就会到一个shard

就不会根据id=1的employee doc的id进行路由了，而是根据parent=1进行路由，会根据父doc的id进行路由，那么就可以通过底层的路由机制，保证父子数据存在于一个shard中

POST /company/employee/\_bulk

{ "index": { "\_id": 2, "parent": "1" }}

{ "name": "李四", "birthday": "1982-05-16", "hobby": "游泳" }

{ "index": { "\_id": 3, "parent": "2" }}

{ "name": "王二", "birthday": "1979-04-01", "hobby": "爬山" }

{ "index": { "\_id": 4, "parent": "3" }}

{ "name": "赵五", "birthday": "1987-05-11", "hobby": "骑马" }

## 根据员工信息和研发中心互相搜索父子数据

已经建立了父子关系的数据模型之后，就要基于这个模型进行各种搜索和聚合

* + - 1. 搜索有1980年以后出生的员工的研发中心

GET /company/rd\_center/\_search

{

"query": {

"has\_child": {

"type": "employee",

"query": {

"range": {

"birthday": {

"gte": "1980-01-01"

}

}

}

}

}

}

{

"took": 33,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 2,

"max\_score": 1,

"hits": [

{

"\_index": "company",

"\_type": "rd\_center",

"\_id": "1",

"\_score": 1,

"\_source": {

"name": "北京研发总部",

"city": "北京",

"country": "中国"

}

},

{

"\_index": "company",

"\_type": "rd\_center",

"\_id": "3",

"\_score": 1,

"\_source": {

"name": "硅谷人工智能实验室",

"city": "硅谷",

"country": "美国"

}

}

]

}

}

* + - 1. 搜索有名叫张三的员工的研发中心

GET /company/rd\_center/\_search

{

"query": {

"has\_child": {

"type": "employee",

"query": {

"match": {

"name": "张三"

}

}

}

}

}

{

"took": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1,

"hits": [

{

"\_index": "company",

"\_type": "rd\_center",

"\_id": "1",

"\_score": 1,

"\_source": {

"name": "北京研发总部",

"city": "北京",

"country": "中国"

}

}

]

}

}

* + - 1. 搜索有至少2个以上员工的研发中心

GET /company/rd\_center/\_search

{

"query": {

"has\_child": {

"type": "employee",

"min\_children": 2,

"query": {

"match\_all": {}

}

}

}

}

{

"took": 5,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1,

"hits": [

{

"\_index": "company",

"\_type": "rd\_center",

"\_id": "1",

"\_score": 1,

"\_source": {

"name": "北京研发总部",

"city": "北京",

"country": "中国"

}

}

]

}

}

* + - 1. 搜索在中国的研发中心的员工

GET /company/employee/\_search

{

"query": {

"has\_parent": {

"parent\_type": "rd\_center",

"query": {

"term": {

"country.keyword": "中国"

}

}

}

}

}

{

"took": 5,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 1,

"hits": [

{

"\_index": "company",

"\_type": "employee",

"\_id": "3",

"\_score": 1,

"\_routing": "2",

"\_parent": "2",

"\_source": {

"name": "王二",

"birthday": "1979-04-01",

"hobby": "爬山"

}

},

{

"\_index": "company",

"\_type": "employee",

"\_id": "1",

"\_score": 1,

"\_routing": "1",

"\_parent": "1",

"\_source": {

"name": "张三",

"birthday": "1970-10-24",

"hobby": "爬山"

}

},

{

"\_index": "company",

"\_type": "employee",

"\_id": "2",

"\_score": 1,

"\_routing": "1",

"\_parent": "1",

"\_source": {

"name": "李四",

"birthday": "1982-05-16",

"hobby": "游泳"

}

}

]

}

}

## 对每个国家的员工兴趣爱好进行聚合统计

统计每个国家的喜欢每种爱好的员工有多少个

GET /company/rd\_center/\_search

{

"size": 0,

"aggs": {

"group\_by\_country": {

"terms": {

"field": "country.keyword"

},

"aggs": {

"group\_by\_child\_employee": {

"children": {

"type": "employee"

},

"aggs": {

"group\_by\_hobby": {

"terms": {

"field": "hobby.keyword"

}

}

}

}

}

}

}

}

{

"took": 15,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_country": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "中国",

"doc\_count": 2,

"group\_by\_child\_employee": {

"doc\_count": 3,

"group\_by\_hobby": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "爬山",

"doc\_count": 2

},

{

"key": "游泳",

"doc\_count": 1

}

]

}

}

},

{

"key": "美国",

"doc\_count": 1,

"group\_by\_child\_employee": {

"doc\_count": 1,

"group\_by\_hobby": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "骑马",

"doc\_count": 1

}

]

}

}

}

]

}

}

}

## 祖孙三层数据关系建模以及搜索实战

父子关系，祖孙三层关系的数据建模，搜索

PUT /company

{

"mappings": {

"country": {},

"rd\_center": {

"\_parent": {

"type": "country"

}

},

"employee": {

"\_parent": {

"type": "rd\_center"

}

}

}

}

country -> rd\_center -> employee，祖孙三层数据模型

POST /company/country/\_bulk

{ "index": { "\_id": "1" }}

{ "name": "中国" }

{ "index": { "\_id": "2" }}

{ "name": "美国" }

POST /company/rd\_center/\_bulk

{ "index": { "\_id": "1", "parent": "1" }}

{ "name": "北京研发总部" }

{ "index": { "\_id": "2", "parent": "1" }}

{ "name": "上海研发中心" }

{ "index": { "\_id": "3", "parent": "2" }}

{ "name": "硅谷人工智能实验室" }

PUT /company/employee/1?parent=1&routing=1

{

"name": "张三",

"dob": "1970-10-24",

"hobby": "爬山"

}

routing参数的讲解，必须跟grandparent相同，否则有问题

country，用的是自己的id去路由; rd\_center，parent，用的是country的id去路由; employee，如果也是仅仅指定一个parent，那么用的是rd\_center的id去路由，这就导致祖孙三层数据不会在一个shard上

孙子辈儿，要手动指定routing，指定为爷爷辈儿的数据的id

搜索有爬山爱好的员工所在的国家

GET /company/country/\_search

{

"query": {

"has\_child": {

"type": "rd\_center",

"query": {

"has\_child": {

"type": "employee",

"query": {

"match": {

"hobby": "爬山"

}

}

}

}

}

}

}

{

"took": 10,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1,

"hits": [

{

"\_index": "company",

"\_type": "country",

"\_id": "1",

"\_score": 1,

"\_source": {

"name": "中国"

}

}

]

}

}

## 基于term vector深入探查数据的情况

* + - 1. term vector介绍

获取document中的某个field内的各个term的统计信息

term information: term frequency in the field, term positions, start and end offsets, term payloads

term statistics: 设置term\_statistics=true; total term frequency, 一个term在所有document中出现的频率; document frequency，有多少document包含这个term

field statistics: document count，有多少document包含这个field; sum of document frequency，一个field中所有term的df之和; sum of total term frequency，一个field中的所有term的tf之和

GET /twitter/tweet/1/\_termvectors

GET /twitter/tweet/1/\_termvectors?fields=text

term statistics和field statistics并不精准，不会被考虑有的doc可能被删除了

我告诉大家，其实很少用，用的时候，一般来说，就是你需要对一些数据做探查的时候。比如说，你想要看到某个term，某个词条，大话西游，这个词条，在多少个document中出现了。或者说某个field，film\_desc，电影的说明信息，有多少个doc包含了这个说明信息。

* + - 1. index-iime term vector实验

term vector，涉及了很多的term和field相关的统计信息，有两种方式可以采集到这个统计信息

* 1. index-time，你在mapping里配置一下，然后建立索引的时候，就直接给你生成这些term和field的统计信息了
  2. query-time，你之前没有生成过任何的Term vector信息，然后在查看term vector的时候，直接就可以看到了，会on the fly，现场计算出各种统计信息，然后返回给你

这一讲，不会手敲任何命令，直接copy我做好的命令，因为这一讲的重点，不是掌握什么搜索或者聚合的语法，而是说，掌握，如何采集term vector信息，然后如何看懂term vector信息，你能掌握利用term vector进行数据探查

PUT /my\_index

{

"mappings": {

"my\_type": {

"properties": {

"text": {

"type": "text",

"term\_vector": "with\_positions\_offsets\_payloads",

"store" : true,

"analyzer" : "fulltext\_analyzer"

},

"fullname": {

"type": "text",

"analyzer" : "fulltext\_analyzer"

}

}

}

},

"settings" : {

"index" : {

"number\_of\_shards" : 1,

"number\_of\_replicas" : 0

},

"analysis": {

"analyzer": {

"fulltext\_analyzer": {

"type": "custom",

"tokenizer": "whitespace",

"filter": [

"lowercase",

"type\_as\_payload"

]

}

}

}

}

}

PUT /my\_index/my\_type/1

{

"fullname" : "Leo Li",

"text" : "hello test test test "

}

PUT /my\_index/my\_type/2

{

"fullname" : "Leo Li",

"text" : "other hello test ..."

}

GET /my\_index/my\_type/1/\_termvectors

{

"fields" : ["text"],

"offsets" : true,

"payloads" : true,

"positions" : true,

"term\_statistics" : true,

"field\_statistics" : true

}

{

"\_index": "my\_index",

"\_type": "my\_type",

"\_id": "1",

"\_version": 1,

"found": true,

"took": 10,

"term\_vectors": {

"text": {

"field\_statistics": {

"sum\_doc\_freq": 6,

"doc\_count": 2,

"sum\_ttf": 8

},

"terms": {

"hello": {

"doc\_freq": 2,

"ttf": 2,

"term\_freq": 1,

"tokens": [

{

"position": 0,

"start\_offset": 0,

"end\_offset": 5,

"payload": "d29yZA=="

}

]

},

"test": {

"doc\_freq": 2,

"ttf": 4,

"term\_freq": 3,

"tokens": [

{

"position": 1,

"start\_offset": 6,

"end\_offset": 10,

"payload": "d29yZA=="

},

{

"position": 2,

"start\_offset": 11,

"end\_offset": 15,

"payload": "d29yZA=="

},

{

"position": 3,

"start\_offset": 16,

"end\_offset": 20,

"payload": "d29yZA=="

}

]

}

}

}

}

}

* + - 1. query-time term vector实验

GET /my\_index/my\_type/1/\_termvectors

{

"fields" : ["fullname"],

"offsets" : true,

"positions" : true,

"term\_statistics" : true,

"field\_statistics" : true

}

一般来说，如果条件允许，你就用query time的term vector就可以了，你要探查什么数据，现场去探查一下就好了

* + - 1. 手动指定doc的term vector

GET /my\_index/my\_type/\_termvectors

{

"doc" : {

"fullname" : "Leo Li",

"text" : "hello test test test"

},

"fields" : ["text"],

"offsets" : true,

"payloads" : true,

"positions" : true,

"term\_statistics" : true,

"field\_statistics" : true

}

手动指定一个doc，实际上不是要指定doc，而是要指定你想要安插的词条，hello test，那么就可以放在一个field中

将这些term分词，然后对每个term，都去计算它在现有的所有doc中的一些统计信息

这个挺有用的，可以让你手动指定要探查的term的数据情况，你就可以指定探查“大话西游”这个词条的统计信息

* + - 1. 手动指定analyzer来生成term vector

GET /my\_index/my\_type/\_termvectors

{

"doc" : {

"fullname" : "Leo Li",

"text" : "hello test test test"

},

"fields" : ["text"],

"offsets" : true,

"payloads" : true,

"positions" : true,

"term\_statistics" : true,

"field\_statistics" : true,

"per\_field\_analyzer" : {

"text": "standard"

}

}

* + - 1. terms filter

GET /my\_index/my\_type/\_termvectors

{

"doc" : {

"fullname" : "Leo Li",

"text" : "hello test test test"

},

"fields" : ["text"],

"offsets" : true,

"payloads" : true,

"positions" : true,

"term\_statistics" : true,

"field\_statistics" : true,

"filter" : {

"max\_num\_terms" : 3,

"min\_term\_freq" : 1,

"min\_doc\_freq" : 1

}

}

这个就是说，根据term统计信息，过滤出你想要看到的term vector统计结果

也挺有用的，比如你探查数据把，可以过滤掉一些出现频率过低的term，就不考虑了

* + - 1. multi term vector

GET \_mtermvectors

{

"docs": [

{

"\_index": "my\_index",

"\_type": "my\_type",

"\_id": "2",

"term\_statistics": true

},

{

"\_index": "my\_index",

"\_type": "my\_type",

"\_id": "1",

"fields": [

"text"

]

}

]

}

GET /my\_index/\_mtermvectors

{

"docs": [

{

"\_type": "test",

"\_id": "2",

"fields": [

"text"

],

"term\_statistics": true

},

{

"\_type": "test",

"\_id": "1"

}

]

}

GET /my\_index/my\_type/\_mtermvectors

{

"docs": [

{

"\_id": "2",

"fields": [

"text"

],

"term\_statistics": true

},

{

"\_id": "1"

}

]

}

GET /\_mtermvectors

{

"docs": [

{

"\_index": "my\_index",

"\_type": "my\_type",

"doc" : {

"fullname" : "Leo Li",

"text" : "hello test test test"

}

},

{

"\_index": "my\_index",

"\_type": "my\_type",

"doc" : {

"fullname" : "Leo Li",

"text" : "other hello test ..."

}

}

]

}

## 深入剖析搜索结果的highlight高亮显示

1. 一个最基本的高亮例子

PUT /blog\_website

{

"mappings": {

"blogs": {

"properties": {

"title": {

"type": "text",

"analyzer": "ik\_max\_word"

},

"content": {

"type": "text",

"analyzer": "ik\_max\_word"

}

}

}

}

}

PUT /blog\_website/blogs/1

{

"title": "我的第一篇博客",

"content": "大家好，这是我写的第一篇博客，特别喜欢这个博客网站！！！"

}

GET /blog\_website/blogs/\_search

{

"query": {

"match": {

"title": "博客"

}

},

"highlight": {

"fields": {

"title": {}

}

}

}

{

"took": 103,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 0.28582606,

"hits": [

{

"\_index": "blog\_website",

"\_type": "blogs",

"\_id": "1",

"\_score": 0.28582606,

"\_source": {

"title": "我的第一篇博客",

"content": "大家好，这是我写的第一篇博客，特别喜欢这个博客网站！！！"

},

"highlight": {

"title": [

"我的第一篇<em>博客</em>"

]

}

}

]

}

}

<em></em>表现，会变成红色，所以说你的指定的field中，如果包含了那个搜索词的话，就会在那个field的文本中，对搜索词进行红色的高亮显示

GET /blog\_website/blogs/\_search

{

"query": {

"bool": {

"should": [

{

"match": {

"title": "博客"

}

},

{

"match": {

"content": "博客"

}

}

]

}

},

"highlight": {

"fields": {

"title": {},

"content": {}

}

}

}

highlight中的field，必须跟query中的field一一对齐的

1. 三种highlight介绍

plain highlight，lucene highlight，默认

posting highlight，index\_options=offsets

* 1. 性能比plain highlight要高，因为不需要重新对高亮文本进行分词
  2. 对磁盘的消耗更少
  3. 将文本切割为句子，并且对句子进行高亮，效果更好

PUT /blog\_website

{

"mappings": {

"blogs": {

"properties": {

"title": {

"type": "text",

"analyzer": "ik\_max\_word"

},

"content": {

"type": "text",

"analyzer": "ik\_max\_word",

"index\_options": "offsets"

}

}

}

}

}

index\_option:offsets设置为offset使用的是posting

highlight

PUT /blog\_website/blogs/1

{

"title": "我的第一篇博客",

"content": "大家好，这是我写的第一篇博客，特别喜欢这个博客网站！！！"

}

GET /blog\_website/blogs/\_search

{

"query": {

"match": {

"content": "博客"

}

},

"highlight": {

"fields": {

"content": {}

}

}

}

fast vector highlight

term vector设置在mapping中，就会用fast verctor highlight

* + - * 1. 对大field而言（大于1mb），性能更高

PUT /blog\_website

{

"mappings": {

"blogs": {

"properties": {

"title": {

"type": "text",

"analyzer": "ik\_max\_word"

},

"content": {

"type": "text",

"analyzer": "ik\_max\_word",

"term\_vector" : "with\_positions\_offsets"

}

}

}

}

}

强制使用某种highlighter，比如对于开启了term vector的field而言，可以强制使用plain highlight

GET /blog\_website/blogs/\_search

{

"query": {

"match": {

"content": "博客"

}

},

"highlight": {

"fields": {

"content": {

"type": "plain"

}

}

}

}

总结一下，其实可以根据你的实际情况去考虑，一般情况下，用plain highlight也就足够了，不需要做其他额外的设置

如果对高亮的性能要求很高，可以尝试启用posting highlight

如果field的值特别大，超过了1M，那么可以用fast vector highlight

1. 设置高亮html标签，默认是<em>标签

GET /blog\_website/blogs/\_search

{

"query": {

"match": {

"content": "博客"

}

},

"highlight": {

"pre\_tags": ["<tag1>"],

"post\_tags": ["</tag2>"],

"fields": {

"content": {

"type": "plain"

}

}

}

}

1. 高亮片段fragment的设置

GET /\_search

{

"query" : {

"match": { "user": "kimchy" }

},

"highlight" : {

"fields" : {

"content" : {"fragment\_size" : 150, "number\_of\_fragments" : 3, "no\_match\_size": 150 }

}

}

}

fragment\_size: 你一个Field的值，比如有长度是1万，但是你不可能在页面上显示这么长啊。。。设置要显示出来的fragment文本判断的长度，默认是100

number\_of\_fragments：你可能你的高亮的fragment文本片段有多个片段，你可以指定就显示几个片段

## 使用search template将搜索模板化

搜索模板，search template，高级功能，就可以将我们的一些搜索进行模板化，然后的话，每次执行这个搜索，就直接调用模板，给传入一些参数就可以了

越高级的功能，越少使用，可能只有在你真的遇到特别合适的场景的时候，才会去使用某个高级功能。

1. search template入门

GET /blog\_website/blogs/\_search/template

{

"inline" : {

"query": {

"match" : {

"{{field}}" : "{{value}}"

}

}

},

"params" : {

"field" : "title",

"value" : "博客"

}

}

GET /blog\_website/blogs/\_search

{

"query": {

"match" : {

"title" : "博客"

}

}

}

search template："{{field}}" : "{{value}}"

1. toJson

GET /blog\_website/blogs/\_search/template

{

"inline": "{\"query\": {\"match\": {{#toJson}}matchCondition{{/toJson}}}}",

"params": {

"matchCondition": {

"title": "博客"

}

}

}

GET /blog\_website/blogs/\_search

{

"query": {

"match" : {

"title" : "博客"

}

}

}

1. join

GET /blog\_website/blogs/\_search/template

{

"inline": {

"query": {

"match": {

"title": "{{#join delimiter=' '}}titles{{/join delimiter=' '}}"

}

}

},

"params": {

"titles": ["博客", "网站"]

}

}

博客,网站

GET /blog\_website/blogs/\_search

{

"query": {

"match" : {

"title" : "博客 网站"

}

}

}

1. default value

POST /blog\_website/blogs/1/\_update

{

"doc": {

"views": 5

}

}

GET /blog\_website/blogs/\_search/template

{

"inline": {

"query": {

"range": {

"views": {

"gte": "{{start}}",

"lte": "{{end}}{{^end}}20{{/end}}"

}

}

}

},

"params": {

"start": 1,

"end": 10

}

}

GET /blog\_website/blogs/\_search

{

"query": {

"range": {

"views": {

"gte": 1,

"lte": 10

}

}

}

}

GET /blog\_website/blogs/\_search/template

{

"inline": {

"query": {

"range": {

"views": {

"gte": "{{start}}",

"lte": "{{end}}{{^end}}20{{/end}}"

}

}

}

},

"params": {

"start": 1

}

}

GET /blog\_website/blogs/\_search

{

"query": {

"range": {

"views": {

"gte": 1,

"lte": 20

}

}

}

}

1. conditional

es的config/scripts目录下，预先保存这个复杂的模板，后缀名是.mustache，文件名是conditonal

{

"query": {

"bool": {

"must": {

"match": {

"line": "{{text}}"

}

},

"filter": {

{{#line\_no}}

"range": {

"line\_no": {

{{#start}}

"gte": "{{start}}"

{{#end}},{{/end}}

{{/start}}

{{#end}}

"lte": "{{end}}"

{{/end}}

}

}

{{/line\_no}}

}

}

}

}

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

{

"took": 4,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1,

"hits": [

{

"\_index": "my\_index",

"\_type": "my\_type",

"\_id": "1",

"\_score": 1,

"\_source": {

"line": "我的博客",

"line\_no": 5

}

}

]

}

}

GET /my\_index/my\_type/\_search/template

{

"file": "conditional",

"params": {

"text": "博客",

"line\_no": true,

"start": 1,

"end": 10

}

}

1. 保存search template

config/scripts，.mustache

提供一个思路

比如说，一般在大型的团队中，可能不同的人，都会想要执行一些类似的搜索操作

这个时候，有一些负责底层运维的一些同学，就可以基于search template，封装一些模板出来，然后是放在各个es进程的scripts目录下的

其他的团队，其实就不用各个团队自己反复手写复杂的通用的查询语句了，直接调用某个搜索模板，传入一些参数就好了

## 基于completion suggest实现搜索提示

suggest，completion suggest，自动完成，搜索推荐，搜索提示 --> 自动完成，auto completion

auto completion

比如说我们在百度，搜索，你现在搜索“大话西游” -->

百度，自动给你提示，“大话西游电影”，“大话西游小说”， “大话西游手游”

不用你把所有你想要输入的文本都输入完，搜索引擎会自动提示你可能是你想要搜索的那个文本

PUT /news\_website

{

"mappings": {

"news" : {

"properties" : {

"title" : {

"type": "text",

"analyzer": "ik\_max\_word",

"fields": {

"suggest" : {

"type" : "completion",

"analyzer": "ik\_max\_word"

}

}

},

"content": {

"type": "text",

"analyzer": "ik\_max\_word"

}

}

}

}

}

completion，es实现的时候，是非常高性能的，会构建不是倒排索引，也不是正拍索引，就是纯的用于进行前缀搜索的一种特殊的数据结构，而且会全部放在内存中，所以auto completion进行的前缀搜索提示，性能是非常高的

大话西游

PUT /news\_website/news/1

{

"title": "大话西游电影",

"content": "大话西游的电影时隔20年即将在2017年4月重映"

}

PUT /news\_website/news/2

{

"title": "大话西游小说",

"content": "某知名网络小说作家已经完成了大话西游同名小说的出版"

}

PUT /news\_website/news/3

{

"title": "大话西游手游",

"content": "网易游戏近日出品了大话西游经典IP的手游，正在火爆内测中"

}

GET /news\_website/news/\_search

{

"suggest": {

"my-suggest" : {

"prefix" : "大话西游",

"completion" : {

"field" : "title.suggest"

}

}

}

}

{

"took": 6,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 0,

"max\_score": 0,

"hits": []

},

"suggest": {

"my-suggest": [

{

"text": "大话西游",

"offset": 0,

"length": 4,

"options": [

{

"text": "大话西游小说",

"\_index": "news\_website",

"\_type": "news",

"\_id": "2",

"\_score": 1,

"\_source": {

"title": "大话西游小说",

"content": "某知名网络小说作家已经完成了大话西游同名小说的出版"

}

},

{

"text": "大话西游手游",

"\_index": "news\_website",

"\_type": "news",

"\_id": "3",

"\_score": 1,

"\_source": {

"title": "大话西游手游",

"content": "网易游戏近日出品了大话西游经典IP的手游，正在火爆内测中"

}

},

{

"text": "大话西游电影",

"\_index": "news\_website",

"\_type": "news",

"\_id": "1",

"\_score": 1,

"\_source": {

"title": "大话西游电影",

"content": "大话西游的电影时隔20年即将在2017年4月重映"

}

}

]

}

]

}

}

GET /news\_website/news/\_search

{

"query": {

"match": {

"content": "大话西游电影"

}

}

}

{

"took": 9,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 1.3495269,

"hits": [

{

"\_index": "news\_website",

"\_type": "news",

"\_id": "1",

"\_score": 1.3495269,

"\_source": {

"title": "大话西游电影",

"content": "大话西游的电影时隔20年即将在2017年4月重映"

}

},

{

"\_index": "news\_website",

"\_type": "news",

"\_id": "3",

"\_score": 1.217097,

"\_source": {

"title": "大话西游手游",

"content": "网易游戏近日出品了大话西游经典IP的手游，正在火爆内测中"

}

},

{

"\_index": "news\_website",

"\_type": "news",

"\_id": "2",

"\_score": 1.1299736,

"\_source": {

"title": "大话西游小说",

"content": "某知名网络小说作家已经完成了大话西游同名小说的出版"

}

}

]

}

}

## 使用动态映射模板定制自己的映射策略

高级的用法

比如说，我们本来没有某个type，或者没有某个field，但是希望在插入数据的时候，es自动为我们做一个识别，动态映射出这个type的mapping，包括每个field的数据类型，一般用的动态映射，dynamic mapping

这里有个问题，如果说，我们其实对dynamic mapping有一些自己独特的需求，比如说，es默认来说，如经过识别到一个数字，field: 10，默认是搞成这个field的数据类型是long，再比如说，如果我们弄了一个field : "10"，默认就是text，还会带一个keyword的内置field。我们没法改变。

但是我们现在就是希望动态映射的时候，根据我们的需求去映射，而不是让es自己按照默认的规则去玩儿

dyanmic mapping template，动态映射模板

我们自己预先定义一个模板，然后插入数据的时候，相关的field，如果能够根据我们预先定义的规则，匹配上某个我们预定义的模板，那么就会根据我们的模板来进行mapping，决定这个Field的数据类型

0、默认的动态映射的效果咋样

DELETE /my\_index

PUT /my\_index/my\_type/1

{

"test\_string": "hello world",

"test\_number": 10

}

es的自动的默认的，动态映射是咋样的。。。

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

{

"my\_index": {

"mappings": {

"my\_type": {

"properties": {

"test\_number": {

"type": "long"

},

"test\_string": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

}

}

}

}

这个就是es的默认的动态映射规则，可能就不是我们想要的。。。

我们比如说，现在想要的效果是啥。。。

test\_number，如果是个数字，我们希望默认就是integer类型的

test\_string，如果是字符串，我们希望默认是个text，这个没问题，但是内置的field名字，叫做raw，不叫做keyword，类型还是keyword，保留500个字符

1. 根据类型匹配映射模板

动态映射模板，有两种方式，第一种，是根据新加入的field的默认的数据类型，来进行匹配，匹配上某个预定义的模板；第二种，是根据新加入的field的名字，去匹配预定义的名字，或者去匹配一个预定义的通配符，然后匹配上某个预定义的模板

PUT my\_index

{

"mappings": {

"my\_type": {

"dynamic\_templates": [

{

"integers": {

"match\_mapping\_type": "long",

"mapping": {

"type": "integer"

}

}

},

{

"strings": {

"match\_mapping\_type": "string",

"mapping": {

"type": "text",

"fields": {

"raw": {

"type": "keyword",

"ignore\_above": 500

}

}

}

}

}

]

}

}

}

PUT /my\_index/my\_type/1

{

"test\_long": 1,

"test\_string": "hello world"

}

{

"my\_index": {

"mappings": {

"my\_type": {

"dynamic\_templates": [

{

"integers": {

"match\_mapping\_type": "long",

"mapping": {

"type": "integer"

}

}

},

{

"strings": {

"match\_mapping\_type": "string",

"mapping": {

"fields": {

"raw": {

"ignore\_above": 500,

"type": "keyword"

}

},

"type": "text"

}

}

}

],

"properties": {

"test\_number": {

"type": "integer"

},

"test\_string": {

"type": "text",

"fields": {

"raw": {

"type": "keyword",

"ignore\_above": 500

}

}

}

}

}

}

}

1. 根据字段名配映射模板

PUT /my\_index

{

"mappings": {

"my\_type": {

"dynamic\_templates": [

{

"string\_as\_integer": {

"match\_mapping\_type": "string",

"match": "long\_\*",

"unmatch": "\*\_text",

"mapping": {

"type": "integer"

}

}

}

]

}

}

}

举个例子，field : "10"，把类似这种field，弄成long型

{

"my\_index": {

"mappings": {

"my\_type": {

"dynamic\_templates": [

{

"string\_as\_integer": {

"match": "long\_\*",

"unmatch": "\*\_text",

"match\_mapping\_type": "string",

"mapping": {

"type": "integer"

}

}

}

],

"properties": {

"long\_field": {

"type": "integer"

},

"long\_field\_text": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

}

}

}

}

场景，有些时候，dynamic mapping + template，每天有一堆日志，每天有一堆数据

这些数据，每天的数据都放一个新的type中，每天的数据都会哗哗的往新的type中写入，此时你就可以定义一个模板，搞一个脚本，每天都预先生成一个新type的模板，里面讲你的各个Field都匹配到一个你预定义的模板中去，就好了

## 使用geo point地理位置数据类型

主要是es支持基于地理位置的搜索，和聚合分析的

举个例子，比如说，我们后面就会给大家演示一下，你现在如果说做了一个酒店o2o app，让你的用户在任何地方，都可以根据当前所在的位置，找到自己身边的符合条件的一些酒店，那么此时就完全可以使用es来实现，非常合适

我现在在上海某个大厦附近，我要搜索到距离我2公里以内的5星级的带游泳池的一个酒店s，用es就完全可以实现类似这样的基于地理位置的搜索引擎

1. 建立geo\_point类型的mapping

第一个地理位置的数据类型，就是geo\_point，geo\_point，说白了，就是一个地理位置坐标点，包含了一个经度，一个维度，经纬度，就可以唯一定位一个地球上的坐标

PUT /my\_index

{

"mappings": {

"my\_type": {

"properties": {

"location": {

"type": "geo\_point"

}

}

}

}

}

1. 写入geo\_point的3种方法

PUT my\_index/my\_type/1

{

"text": "Geo-point as an object",

"location": {

"lat": 41.12,

"lon": -71.34

}

}

latitude：维度

longitude：经度

我们这里就不用去关心，这些坐标到底代表什么地方，其实都是我自己随便写的，只要能够作为课程，给大家演示清楚就可以了，自己去找一些提供地理位置的一些公司，供应商，api，百度地图，也是提供各个地方的经纬度的

不建议用下面两种语法

PUT my\_index/my\_type/2

{

"text": "Geo-point as a string",

"location": "41.12,-71.34"

}

PUT my\_index/my\_type/4

{

"text": "Geo-point as an array",

"location": [ -71.34, 41.12 ]

}

1. 根据地理位置进行查询

最最简单的，根据地理位置查询一些点，比如说，下面geo\_bounding\_box查询，查询某个矩形的地理位置范围内的坐标点

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

{

"query": {

"geo\_bounding\_box": {

"location": {

"top\_left": {

"lat": 42,

"lon": -72

},

"bottom\_right": {

"lat": 40,

"lon": -74

}

}

}

}

}

{

"took": 81,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1,

"hits": [

{

"\_index": "my\_index",

"\_type": "my\_type",

"\_id": "1",

"\_score": 1,

"\_source": {

"location": {

"lat": 41.12,

"lon": -71.34

}

}

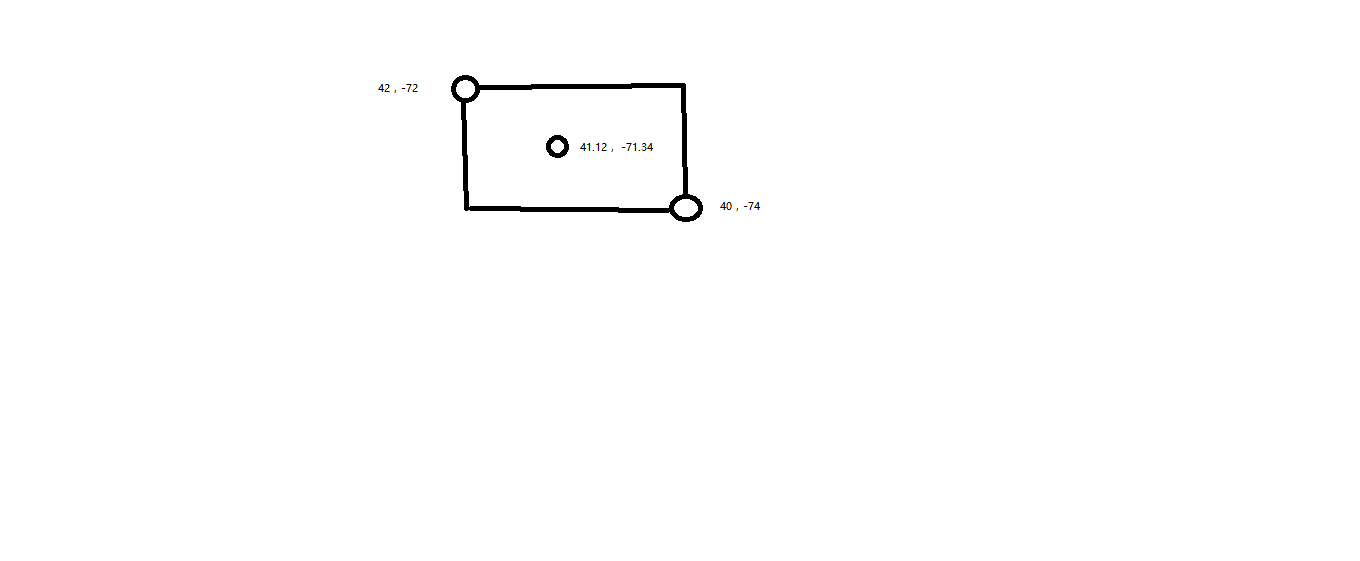
}

]

}

}

比如41.12,-71.34就是一个酒店，然后我们现在搜索的是从42,-72（代表了大厦A）和40,-74（代表了马路B）作为矩形的范围，在这个范围内的酒店，是什么

**地理位置**

## 酒店o2o搜索案例以及搜索指定区域内的酒店

酒店o2o app作为一个背景，用各种各样的方式，去搜索你当前所在的地理位置附近的酒店

搜索指定区域范围内的酒店，比如说，我们可以在搜索的时候，指定两个地点，就要在东方明珠大厦和上海路组成的矩阵的范围内，搜索我想要的酒店

PUT /hotel\_app

{

"mappings": {

"hotels": {

"properties": {

"pin": {

"properties": {

"location": {

"type": "geo\_point"

}

}

}

}

}

}

}

PUT /hotel\_app/hotels/1

{

"name": "喜来登大酒店",

"pin" : {

"location" : {

"lat" : 40.12,

"lon" : -71.34

}

}

}

GET /hotel\_app/hotels/\_search

{

"query": {

"bool": {

"must": [

{

"match\_all": {}

}

],

"filter": {

"geo\_bounding\_box": {

"pin.location": {

"top\_left" : {

"lat" : 40.73,

"lon" : -74.1

},

"bottom\_right" : {

"lat" : 40.01,

"lon" : -71.12

}

}

}

}

}

}

}

GET /hotel\_app/hotels/\_search

{

"query": {

"bool": {

"must": [

{

"match\_all": {}

}

],

"filter": {

"geo\_polygon": {

"pin.location": {

"points": [

{"lat" : 40.73, "lon" : -74.1},

{"lat" : 40.01, "lon" : -71.12},

{"lat" : 50.56, "lon" : -90.58}

]

}

}

}

}

}

}

我们现在要指定东方明珠大厦，上海路，上海博物馆，这三个地区组成的多边形的范围内，我要搜索这里面的酒店

## 搜索距离当前位置一定范围内的酒店

酒店o2o app，作为案例背景

比如说，现在用户，所在的位置，是个地理位置的坐标，我是知道我的坐标的，app是知道的，android，地理位置api，都可以拿到当前手机app的经纬度

我说，我现在就要搜索出，举例我200m，或者1公里内的酒店

重要！！！！

我们之前出去玩儿，都会用一些酒店o2o app，典型的代表，很多旅游app，一般来说，我们怎么搜索，到了一个地方，就会搜索说，我这里举例几百米，2公里内的酒店，搜索一下

上节课讲解的，其实也很重要，一般来说，发生在我们在去旅游之前，会现在旅游app上搜索一个区域内的酒店，比如说，指定了西安火车站、西安博物馆，拿指定的几个地方的地理位置，组成一个多边形区域范围，去搜索这个区域内的酒店

承认，一些案例，当然不可能说达到讲解真实的复杂的大型的项目的效果来的那么好，光是学知识，学技术而言，有一些案例就非常不错了

后面，会讲解真正的企业级的大型的搜索引擎，真实复杂业务的数据分析系统的项目

GET /hotel\_app/hotels/\_search

{

"query": {

"bool": {

"must": [

{

"match\_all": {}

}

],

"filter": {

"geo\_distance": {

"distance": "200km",

"pin.location": {

"lat": 40,

"lon": -70

}

}

}

}

}

}

## 统计当前位置每个距离范围内有多少家酒店

基于地理位置进行聚合分析

需求就是，统计一下，举例我当前坐标的几个范围内的酒店的数量，比如说举例我0~100m有几个酒店，100m~300m有几个酒店，300m以上有几个酒店

一般来说，做酒店app，一般来说，我们是不是会有一个地图，用户可以在地图上直接查看和搜索酒店，此时就可以显示出来举例你当前的位置，几个举例范围内，有多少家酒店，让用户知道，心里清楚，用户体验就比较好

GET /hotel\_app/hotels/\_search

{

"size": 0,

"aggs": {

"agg\_by\_distance\_range": {

"geo\_distance": {

"field": "pin.location",

"origin": {

"lat": 40,

"lon": -70

},

"unit": "mi",

"ranges": [

{

"to": 100

},

{

"from": 100,

"to": 300

},

{

"from": 300

}

]

}

}

}

}

{

"took": 5,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 0,

"hits": []

},

"aggregations": {

"agg\_by\_distance\_range": {

"buckets": [

{

"key": "\*-100.0",

"from": 0,

"to": 100,

"doc\_count": 1

},

{

"key": "100.0-300.0",

"from": 100,

"to": 300,

"doc\_count": 0

},

{

"key": "300.0-\*",

"from": 300,

"doc\_count": 0

}

]

}

}

}

m (metres) but it can also accept: m (miles), km (kilometers)

sloppy\_arc (the default), arc (most accurate) and plane (fastest)

## client集群自动探查以及汽车零售店案例背景

1. client集群自动探查

默认情况下，是根据我们手动指定的所有节点，依次轮询这些节点，来发送各种请求的，如下面的代码，我们可以手动为client指定多个节点

TransportClient client = new PreBuiltTransportClient(settings)

.addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("localhost1"), 9300))

.addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("localhost2"), 9300))

.addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("localhost3"), 9300));

但是问题是，如果我们有成百上千个节点呢？难道也要这样手动添加吗？

es client提供了一种集群节点自动探查的功能，打开这个自动探查机制以后，es client会根据我们手动指定的几个节点连接过去，然后通过集群状态自动获取当前集群中的所有data node，然后用这份完整的列表更新自己内部要发送请求的node list。默认每隔5秒钟，就会更新一次node list。

但是注意，es cilent是不会将Master node纳入node list的，因为要避免给master node发送搜索等请求。

这样的话，我们其实直接就指定几个master node，或者1个node就好了，client会自动去探查集群的所有节点，而且每隔5秒还会自动刷新。非常棒。

Settings settings = Settings.builder()

.put("client.transport.sniff", true).build();

TransportClient client = new PreBuiltTransportClient(settings);

使用上述的settings配置，将client.transport.sniff设置为true即可打开集群节点自动探查功能

在实际的生产环境中，都是这么玩儿的。。。

1. 汽车零售案例背景

简单来说，会涉及到三个数据，汽车信息，汽车销售记录，汽车4S店信息

## 基于upsert实现汽车最新价格的调整

做一个汽车零售数据的mapping，我们要做的第一份数据，其实汽车信息

PUT /car\_shop

{

"mappings": {

"cars": {

"properties": {

"brand": {

"type": "text",

"analyzer": "ik\_max\_word",

"fields": {

"raw": {

"type": "keyword"

}

}

},

"name": {

"type": "text",

"analyzer": "ik\_max\_word",

"fields": {

"raw": {

"type": "keyword"

}

}

}

}

}

}

}

首先的话呢，第一次调整宝马320这个汽车的售价，我们希望将售价设置为32万，用一个upsert语法，如果这个汽车的信息之前不存在，那么就insert，如果存在，那么就update

IndexRequest indexRequest = new IndexRequest("car\_shop", "cars", "1")

.source(jsonBuilder()

.startObject()

.field("brand", "宝马")

.field("name", "宝马320")

.field("price", 320000)

.field("produce\_date", "2017-01-01")

.endObject());

UpdateRequest updateRequest = new UpdateRequest("car\_shop", "cars", "1")

.doc(jsonBuilder()

.startObject()

.field("price", 320000)

.endObject())

.upsert(indexRequest);

client.update(updateRequest).get();

IndexRequest indexRequest = new IndexRequest("car\_shop", "cars", "1")

.source(jsonBuilder()

.startObject()

.field("brand", "宝马")

.field("name", "宝马320")

.field("price", 310000)

.field("produce\_date", "2017-01-01")

.endObject());

UpdateRequest updateRequest = new UpdateRequest("car\_shop", "cars", "1")

.doc(jsonBuilder()

.startObject()

.field("price", 310000)

.endObject())

.upsert(indexRequest);

client.update(updateRequest).get();

[代码](基于upsert实现汽车最新价格的调整-代码)

## 基于mget实现多辆汽车的配置与价格对比

场景，一般来说，我们都可以在一些汽车网站上，或者在混合销售多个品牌的汽车4S店的内部，都可以在系统里调出来多个汽车的信息，放在网页上，进行对比

mget，一次性将多个document的数据查询出来，放在一起显示，多个汽车的型号，一次性拿出了多辆汽车的信息

PUT /car\_shop/cars/2

{

"brand": "奔驰",

"name": "奔驰C200",

"price": 350000,

"produce\_date": "2017-01-05"

}

MultiGetResponse multiGetItemResponses = client.prepareMultiGet()

.add("car\_shop", "cars", "1")

.add("car\_shop", "cars", "2")

.get();

for (MultiGetItemResponse itemResponse : multiGetItemResponses) {

GetResponse response = itemResponse.getResponse();

if (response.isExists()) {

String json = response.getSourceAsString();

}

}

[代码](基于mget实现多辆汽车的配置与价格对比-代码)

## 基于bulk实现多4S店销售数据批量上传

业务场景：有一个汽车销售公司，拥有很多家4S店，这些4S店的数据，都会在一段时间内陆续传递过来，汽车的销售数据，现在希望能够在内存中缓存比如1000条销售数据，然后一次性批量上传到es中去

PUT /car\_shop/sales/1

{

"brand": "宝马",

"name": "宝马320",

"price": 320000,

"produce\_date": "2017-01-01",

"sale\_price": 300000,

"sale\_date": "2017-01-21"

}

PUT /car\_shop/sales/2

{

"brand": "宝马",

"name": "宝马320",

"price": 320000,

"produce\_date": "2017-01-01",

"sale\_price": 300000,

"sale\_date": "2017-01-21"

}

BulkRequestBuilder bulkRequest = client.prepareBulk();

bulkRequest.add(client.prepareIndex("car\_shop", "sales", "3")

.setSource(jsonBuilder()

.startObject()

.field("brand", "奔驰")

.field("name", "奔驰C200")

.field("price", 350000)

.field("produce\_date", "2017-01-05")

.field("sale\_price", 340000)

.field("sale\_date", "2017-02-03")

.endObject()

)

);

bulkRequest.add(client.prepareUpdate("car\_shop", "sales", "1")

.setDoc(jsonBuilder()

.startObject()

.field("sale\_price", "290000")

.endObject()

)

);

bulkRequest.add(client.prepareDelete("car\_shop", "sales", "2"));

BulkResponse bulkResponse = bulkRequest.get();

if (bulkResponse.hasFailures()) {}

[代码](基于bulk实现多4S店销售数据批量上传-代码)

## 基于scroll实现月度销售数据批量下载

比如说，现在要下载大批量的数据，从es，放到excel中，我们说，月度，或者年度，销售记录，很多，比如几千条，几万条，几十万条

其实就要用到我们之前讲解的es scroll api，对大量数据批量的获取和处理

PUT /car\_shop/sales/4

{

"brand": "宝马",

"name": "宝马320",

"price": 320000,

"produce\_date": "2017-01-01",

"sale\_price": 280000,

"sale\_date": "2017-01-25"

}

就是要看宝马的销售记录

2条数据，做一个演示，每个批次下载一条宝马的销售记录，分2个批次给它下载完

SearchResponse scrollResp = client.prepareSearch("car\_shop")

.addTypes("sales")

.setScroll(new TimeValue(60000))

.setQuery(termQuery("brand.raw", "宝马"))

.setSize(1)

.get();

do {

for (SearchHit hit : scrollResp.getHits().getHits()) {

}

scrollResp = client.prepareSearchScroll(scrollResp.getScrollId())

.setScroll(new TimeValue(60000))

.execute()

.actionGet();

} while(scrollResp.getHits().getHits().length != 0);

[代码](基于scroll实现月度销售数据批量下载-代码)

## 基于search template按品牌分页查询模板

搜索模板的功能，java api怎么去调用一个搜索模板

page\_query\_by\_brand.mustache

{

"from": {{from}},

"size": {{size}},

"query": {

"match": {

"brand.keyword": "{{brand}}"

}

}

}

SearchResponse sr = new SearchTemplateRequestBuilder(client)

.setScript("page\_query\_by\_brand")

.setScriptType(ScriptService.ScriptType.FILE)

.setScriptParams(template\_params)

.setRequest(new SearchRequest())

.get()

.getResponse();

[代码](基于search%20template实现按品牌分页查询模板-代码)

## 对汽车品牌进行全文检索、精准查询和前缀搜索

PUT /car\_shop/cars/5

{

"brand": "华晨宝马",

"name": "宝马318",

"price": 270000,

"produce\_date": "2017-01-20"

}

SearchResponse response = client.prepareSearch("car\_shop")

.setTypes("cars")

.setQuery(QueryBuilders.matchQuery("brand", "宝马"))

.get();

SearchResponse response = client.prepareSearch("car\_shop")

.setTypes("cars")

.setQuery(QueryBuilders.multiMatchQuery("宝马", "brand", "name"))

.get();

SearchResponse response = client.prepareSearch("car\_shop")

.setTypes("cars")

.setQuery(QueryBuilders.commonTermsQuery("name", "宝马320"))

.get();

SearchResponse response = client.prepareSearch("car\_shop")

.setTypes("cars")

.setQuery(QueryBuilders.prefixQuery("name", "宝"))

.get();

[代码](对汽车品牌进行全文检索、精准查询和前缀搜索--代码)

## 对汽车品牌进行多种条件的组合搜索

QueryBuilder qb = boolQuery()

.must(matchQuery("brand", "宝马"))

.mustNot(termQuery("name.raw", "宝马318"))

.should(termQuery("produce\_date", "2017-01-02"))

.filter(rangeQuery("price").gte("280000").lt("350000"));

SearchResponse response = client.prepareSearch("car\_shop")

.setTypes("cars")

.setQuery(qb)

.get();

[代码](对汽车品牌进行多种条件的组合搜索--代码)

## 基于地理位置对周围汽车4S店进行搜索

<dependency>

<groupId>org.locationtech.spatial4j</groupId>

<artifactId>spatial4j</artifactId>

<version>0.6</version>

</dependency>

<dependency>

<groupId>com.vividsolutions</groupId>

<artifactId>jts</artifactId>

<version>1.13</version>

<exclusions>

<exclusion>

<groupId>xerces</groupId>

<artifactId>xercesImpl</artifactId>

</exclusion>

</exclusions>

</dependency>

比如我们有很多的4s店，然后呢给了用户一个app，在某个地方的时候，可以根据当前的地理位置搜索一下，自己附近的4s店

POST /car\_shop/\_mapping/shops

{

"properties": {

"pin": {

"properties": {

"location": {

"type": "geo\_point"

}

}

}

}

}

PUT /car\_shop/shops/1

{

"name": "上海至全宝马4S店",

"pin" : {

"location" : {

"lat" : 40.12,

"lon" : -71.34

}

}

}

第一个需求：搜索两个坐标点组成的一个区域

QueryBuilder qb = geoBoundingBoxQuery("pin.location").setCorners(40.73, -74.1, 40.01, -71.12);

第二个需求：指定一个区域，由三个坐标点，组成，比如上海大厦，东方明珠塔，上海火车站

List<GeoPoint> points = new ArrayList<>();

points.add(new GeoPoint(40.73, -74.1));

points.add(new GeoPoint(40.01, -71.12));

points.add(new GeoPoint(50.56, -90.58));

QueryBuilder qb = geoPolygonQuery("pin.location", points);

第三个需求：搜索距离当前位置在200公里内的4s店

QueryBuilder qb = geoDistanceQuery("pin.location").point(40, -70).distance(200, DistanceUnit.KILOMETERS);

SearchResponse response = client.prepareSearch("car\_shop")

.setTypes("shops")

.setQuery(qb)

.get();

[代码](基于地理位置对周围汽车4S店进行搜索--代码)