Elasticsearch 分享

# 简介

## Lucene是什么？

Lucene可以被认为是迄今为止最先进、性能最好的、功能最全的搜索引擎库。

但是，Lucene只是一个库。

## Elasticsearch是什么？

Elasticsearch是一个基于Apache Lucene(TM)的开源搜索引擎。它使用Lucene作为其核心来实现是实现所有索引和搜索的功能。并且实现了分布式，restful api，集群管理，系统监控，故障恢复等功能。让使用者可以轻松上手使用。

Elasticsearch能做什么？

分布式的实时文件存储，每个字段都被索引并可被搜索

分布式的实时分析搜索引擎

可以扩展到上百台服务器，处理PB级结构化或非结构化数据

# 基础知识

## 面向文档

类比传统关系型数据库：

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

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

### ****索引(indices)****

### ****类型(type)****

### ****文档(documents)：****

ES存储的都是文档，并且索引（index）文档的内容。

文档是支持复杂结构的，并且支持几乎所有的数据类型（但是Map结构需要特别注意）

文档是可以嵌套的

{

"email": "john@smith.com",

"first\_name": "John",

"last\_name": "Smith",

"info": {

"bio": "Eco-warrior and defender of the weak",

"age": 25,

"interests": [ "dolphins", "whales" ]

},

"join\_date": "2014/05/01"

}

### ****字段（Fields）****

文档中的所有属性都是字段

上面的email

嵌套结构中字段表示info.bio

## 搜索

### query

会计算匹配分数

### 过滤器(filter)

不计算匹配分数，可以缓存结果，用来对搜索结果作过滤

### **聚合(aggregations)**

对搜索结果作group by操作，是一种实时数据统计

### Routing

路由，可以直接找到搜素内容在那个分片，而不需要进行集群搜索

# 必须了解的几个es的核心算法

## 倒排

Elasticsearch使用一种叫做**倒排索引(inverted index)**的结构来做快速的全文搜索。倒排索引由在文档中出现的唯一的单词列表，以及对于每个单词在文档中的位置组成。

例如，我们有两个文档，每个文档content字段包含：

1. The quick brown fox jumped over the lazy dog
2. Quick brown foxes leap over lazy dogs in summer

为了创建倒排索引，我们首先切分每个文档的content字段为单独的单词（我们把它们叫做**词(terms)**或者**表征(tokens)**）（译者注：关于terms和tokens的翻译比较生硬，只需知道语句分词后的个体叫做这两个。），把所有的唯一词放入列表并排序，结果是这个样子的：

| **Term** | **Doc\_1** | **Doc\_2** |
| --- | --- | --- |
| Quick |  | X |
| The | X |  |
| brown | X | X |
| dog | X |  |
| dogs |  | X |
| fox | X |  |
| foxes |  | X |
| in |  | X |
| jumped | X |  |
| lazy | X | X |
| leap |  | X |
| over | X | X |
| quick | X |  |
| summer |  | X |
| the | X |  |

现在，如果我们想搜索"quick brown"，我们只需要找到每个词在哪个文档中出现即可：

| **Term** | **Doc\_1** | **Doc\_2** |
| --- | --- | --- |
| brown | X | X |
| quick | X |  |
| ----- | ------- | ----- |
| Total | 2 | 1 |

两个文档都匹配，但是第一个比第二个有更多的匹配项。 如果我们加入简单的**相似度算法(similarity algorithm)**，计算匹配单词的数目，这样我们就可以说第一个文档比第二个匹配度更高——对于我们的查询具有更多相关性。

## Doc\_Value

Doc values 的存在是因为倒排索引只对某些操作是高效的。 倒排索引的优势 在于查找包含某个项的文档，而对于从另外一个方向的相反操作并不高效，即：确定哪些项是否存在单个文档里，聚合需要这种次级的访问模式。

对于以下倒排索引：

Term Doc\_1 Doc\_2 Doc\_3

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

brown | X | X |

dog | X | | X

dogs | | X | X

fox | X | | X

foxes | | X |

in | | X |

jumped | X | | X

lazy | X | X |

leap | | X |

over | X | X | X

quick | X | X | X

summer | | X |

the | X | | X

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

如果我们想要获得所有包含 brown 的文档的词的完整列表，我们会创建如下查询：

GET /my\_index/\_search

{

"query" : {

"match" : {

"body" : "brown"

}

},

"aggs" : {

"popular\_terms": {

"terms" : {

"field" : "body"

}

}

}

}

查询部分简单又高效。倒排索引是根据项来排序的，所以我们首先在词项列表中找到 brown ，然后扫描所有列，找到包含 brown 的文档。我们可以快速看到 Doc\_1 和 Doc\_2 包含 brown 这个 token。

然后，对于聚合部分，我们需要找到 Doc\_1 和 Doc\_2 里所有唯一的词项。 用倒排索引做这件事情代价很高： 我们会迭代索引里的每个词项并收集 Doc\_1 和 Doc\_2 列里面 token。这很慢而且难以扩展：随着词项和文档的数量增加，执行时间也会增加。

Doc values 通过转置两者间的关系来解决这个问题。倒排索引将词项映射到包含它们的文档，doc values 将文档映射到它们包含的词项：

Doc Terms

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

Doc\_1 | brown, dog, fox, jumped, lazy, over, quick, the

Doc\_2 | brown, dogs, foxes, in, lazy, leap, over, quick, summer

Doc\_3 | dog, dogs, fox, jumped, over, quick, the

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

当数据被转置之后，想要收集到 Doc\_1 和 Doc\_2 的唯一 token 会非常容易。获得每个文档行，获取所有的词项，然后求两个集合的并集。

## Stored fields

当你索引数据的时候， elasticsearch 会保存一份源文档到 \_source ，如果文档的某一字段设置了 store 为 yes (默认为 no)，这时候会在 \_source 存储之外再为这个字段独立进行存储。

https://www.elastic.co/guide/en/elasticsearch/client/net-api/5.x/returned-fields.html#\_stored\_fields

# 深入了解es做的那些事

## 一切从Mapping开始

Mapping就是在Es中建立一个文档结构，相当于Relational DB中create table。

Mapping时，你必须清楚地知道哪些字段需要index（索引），哪些字段需要doc\_value，哪些字段需要store。

## 写入索引

Elasticsearch集群中的每个节点都包含了改节点上分片的元数据信息。协调节点(默认)使用文档ID参与计算，以便为路由提供合适的分片。Elasticsearch使用[MurMurHash3](https://en.wikipedia.org/wiki/MurmurHash)函数对文档ID进行哈希，其结果再对分片数量取模，得到的结果即是索引文档的分片。

shard = hash(document\_id) % (num\_of\_primary\_shards)

如果你设置了routing，那么算法会变成：

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

Es的文档就是个json结构的字符串，写入时会根据mapping的结构，存储文档，建立索引，以及分词。

## 接着你要知道搜索是怎么进行的

### Query阶段



任意节点接受到search request，会分发到其他节点，每个节点都会对这次request做query搜索，并且返回docid和排序。

值得注意的是，每个节点都会执行同样的search。

### Fetch阶段



获得request的节点，根据其他节点返回的docid和排序，根据排序和请求需要数据条数，merge数据，并记录docid和node的关系，重新发起fetch请求，其他节点根据fetch请求获得doc，返回给发起节点，发起节点组装数据并返回response给client

当有routing的时候



## 更新(Update)和删除(Delete)的那些必须知道的事

Es的文档是不可变的。

Delete操作不会删除文档，而是把文档标记为删除，搜索时会忽略这些文档

Update操作其实是删除+新增的过程。同样不会删除文档，仅仅是标记为删除。

频繁的U和D操作会产生大量的segment，而segment越多，搜索效率会越低。必须通过optimize优化索引。（我们下面会讲）

Es是相对实时的不是绝对实时的，因为es需要通过一个refresh机制，才能把改变的数据从缓存区写到磁盘上，之后我们才能搜索到最新的数据。

一般refresh默认是1s。当然这个值是可以配置的。通常我们需要在代码中显式执行refresh比较好。

# 用插件改变es

ES的代码十分开放，你可以通过他的plugin modules（插件模块）实现属于自己的处理逻辑。

Es有很多丰富的插件都是基于这个模块实现的。如ik分词器，拼音分词，安全验证插件等等。

简单自定义一个restful的接口插件，es源码下的deletebyquery插件。

实现plugin接口

public class DeleteByQueryPlugin extends Plugin {  
  
 }

实现restHandler接口

public class RestDeleteByQueryAction extends BaseRestHandler

实现tcp接口HandledTransportAction

public class TransportDeleteByQueryAction extends HandledTransportAction<DeleteByQueryRequest, DeleteByQueryResponse>

集群接口的交互用的都是tcp，必须实现。

# [ElasticSearch自定义排序](http://www.dcharm.com/?p=469)

**基于Groovy脚本的自定义排序**

Groovy脚本即可以直接放在config/scripts下面，服务端运行，也可以通过request直接传输groovy脚本执行。

"query": {

"function\_score": {

"query": {

"match": {

"body": "foo"

}

},

"functions": [

{

"script ": {

"lang": "groovy",

"source": "Math.log(2 + doc['likes'].value)"

"params": {

"my\_modifier": 8

}

}

}

]

}

}}'

**基于native脚本的自定义排序**

ES中的native脚本使用java语言编写，并需要实现相应的NativeScriptFactory接口

**Groovy脚本自定义排序和Native自定排序的比较**  
相对于Native自定义排序，Groovy脚本自定义排序实现和部署方便：不用编译打包，不用修改配置文件，也不用重启ES，但是Native自定义排序的速度是Groovy脚本自定义排序速度的1到3倍，因此建议大家使用Groovy脚本做实验，但是在生产环境中使用Native脚本。

# Es实战

## 必须注意的配置

### 节点

* **主节点**：控制Elasticsearch集群，负责集群中的操作，比如创建/删除一个索引，跟踪集群中的节点，分配分片到节点。主节点处理集群的状态并广播到其他节点，并接收其他节点的确认响应。

每个节点都可以通过设定配置文件elasticsearch.yml中的**node.master**属性为**true**(默认)成为主节点。

对于大型的生产集群来说，推荐使用一个专门的主节点来控制集群，该节点将不处理任何用户请求。

* **数据节点**：持有数据和倒排索引。默认情况下，每个节点都可以通过设定配置文件elasticsearch.yml中的**node.data**属性为**true**(默认)成为数据节点。如果我们要使用一个专门的主节点，应将其**node.data**属性设置为**false**。
* **客户端节点**：如果我们将**node.master**属性和**node.data**属性都设置为**false**，那么该节点就是一个客户端节点，扮演一个负载均衡的角色，将到来的请求路由到集群中的各个节点。

### 禁用swap

bootstrap.mlockall: true

设置为 true 来锁住内存。因为当JVM开始 swapping 时elasticsearch的效率会降低，所以要保证它不 swap，可以把 ES\_MIN\_MEM 和 ES\_MAX\_MEM 两个环境变量设置成同一个值，并且保证机器有足够的内存分配给elasticsearch。同时也要允许 elasticsearch 的进程可以锁住内存，linux 下可以通过 `ulimit -l unlimited` 命令。

### 集群配置

discovery.zen.ping.timeout: 3s

设置集群中自动发现其它节点时 ping 连接超时时间，默认为 3 秒，对于比较差的网络环境可以高点的值来防止自动发现时出错。

discovery.zen.ping.multicast.enabled: false

设置是否打开多播发现节点，默认是 true。

discovery.zen.ping.unicast.hosts: ["host1", "host2:port", "host3[portX-portY]"]

设置集群中master节点的初始列表，可通过这些节点来自动发现新加入集群的节点。

### Refresh配置

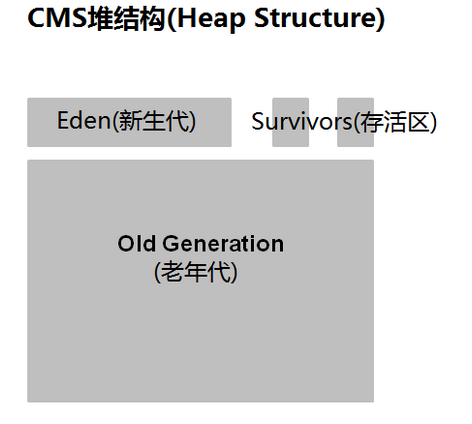
index.refresh\_interval：1s

这样所有新建的索引都使用这个刷新频率。

## GC配置

在bin/elasticsearch.in.sh中进行配置  
修改配置项为尽量大的内存：  
ES\_MIN\_MEM=8g  
ES\_MAX\_MEM=8g  
两者最好改成一样的，否则容易引发长时间GC（stop-the-world）。

不要超过32G

Es默认使用的GC是CMS GC  


建议使用默认配置就好，不要用G1回收，CPU占用太高。

## 监控接口

| **功能** | **命令** |
| --- | --- |
| 查看集群状态 | curl -XGET ‘<http://localhost:9200/_cluster/health?pretty>’ |
| 列出节点信息 | curl -XGET ‘<http://localhost:9200/_cat/nodes?v>’ |
| 列出索引信息 | curl -XGET ‘<http://localhost:9200/_cat/indices?v>’ |
| 列出分片信息 | curl -XGET ‘<http://localhost:9200/_cat/shards?v>’ |
| 查看进程信息 | curl -XGET ‘<http://localhost:9200/_nodes/process?pretty>’ |
| 获取统计信息 | curl -XGET ‘<http://localhost:9200/_stats?pretty>’ |
| 获取统计信息-执行索引 | curl -XGET ‘<http://localhost:9200/productindex/_stats?pretty>’ |
| 热点线程 | curl -XGET ‘<http://localhost:9200/_nodes/hot_threads?pretty>’ |
| 每10sdump热点线程 | curl -XGET ‘<http://localhost:9200/_nodes/hot_threads?type=cpu&interval=10s>’ |
| 获取索引信息 | curl -XGET ‘<http://localhost:9200/productindex/_mapping?pretty>’ |
| 获取JVM信息 | curl -XGET “<http://localhost:9200/_nodes/stats/jvm?pretty>” |
| 获取JVM信息-指定节点 | curl -XGET “<http://localhost:9200/_nodes/yoho.node.114/stats/jvm?pretty>” |
| 删除全部索引 | curl -XDELETE ‘<http://localhost:9200/>\*?pretty’ |
| 删除指定索引 | curl -XDELETE ‘<http://localhost:9200/storagesku?pretty>’ |
| 查看段信息 | curl -XGET ‘<http://localhost:9200/productindex/_segments>’ |
| 执行段合并 | curl -XPOST ‘<http://localhost:9200/productindex/_forcemerge?max_num_segments=1>’ |
| 查看线程池配置 | curl -XGET “<http://localhost:9200/_nodes/thread_pool/>” |

## 案例分析

### Routing使用中的热点问题

现象：

证券股票以市场做Routing，即SH，SZ，IX，FX等市场routing分布到不同shard。

我们假设routing后，node1中存在SH，FX和SZ副本。那么当这3个市场搜索频率升高时，这个node1的searchrequest会大幅上升，成为一个搜索热点node。这可能导致搜索这三个市场的搜索响应变慢。这就是热点问题。

解决方法：

更合理的routing数据，把routing规则细化。

热点问题总是会发生的，只是我们在routing是需要评估热点问题带来的影响。如果影响很大应该更改routing方案或者放弃routing。

### 深度分页问题

在有 5 个主分片的索引中搜索。 当我们请求结果的第一页（结果从 1 到 10 ）



每一个分片产生前10的结果，并且返回给 *协调节点* ，协调节点对 50 个结果排序得到全部结果的前10个。每个分片中的队列如图。这些数据是我们需要的，并且队列长度为10。

现在假设我们请求第 1000 页--结果从 10001 到 10010 。



图中只有红框部分数据是我们需要的。

所有都以相同的方式工作除了每个分片不得不产生前10010个结果以外。 然后协调节点对全部 50050 个结果排序最后丢弃掉这些结果中的 50040 个结果。

可以看到，在分布式系统中，对结果排序的成本随分页的深度成指数上升。

解决方案：尽可能不要使用深度搜索，优化你的算法。

### Segments过多

现象

随着时间的推移，ElasicSearch中每个shard的数据也会越来越多，索引越来越大，而生成的segment（在每个shard中，每个索引文件实际是由多个sgment文件组成）也会越来越多。而segment越多的话，则查询的性能越差，所以通过调用optimize命令，将多个segment合并成更少数量的segment(最少为一个)，从而来提高查询性能。

查看/\_segments?v ，发现索引存在大量segment。

调用命令如下：

http://localhost:9200/indexName/\_optimize?max\_num\_segments=1&wait\_for\_merge=false

参数说明：

1. max\_num\_segments：段数优化。要全面优化索引，将其设置为1。默认设置只需检查是否需要执行一个合并，如果需要，则执行它。【经过测试，该值越小，查询速度越快】
2. only\_expunge\_deletes：该优化操作是否只清空打有删除标签的索引记录。在Lucence中，在执行删除操作时，不会直接删除segment中的记录，而是对该记录打上delete标签。当多个segment进行合并操作时，就会生成一个新的segment，而该新的segment中不再包含删除的记录。这个参数允许只对哪些包含删除记录的segment进行优化操作。
3. Flush：在执行完优化操作之后，再执行刷新操作。默认值为true
4. wait\_for\_merge：当该参数设置为true时，表示其他请求操作要等到合并segment操作结束之后，再进行响应。值得注意的是，由于这个优化操作是一个非常耗时，耗资源的事情，用户提交的请求操作是不能容忍等待这么久，所以这个参数最好设置为false.

### 无用的Query

**现象：**

搜索效率低下

Hotthread监控热点搜索的metric。发现每次搜素都会计算score。

实际上除了分词搜索中会用到score来计算相似度外，其他时候并不需要计算分数。（特殊需求除外）。

**解决方案：**

把无用的query替换成filter。

### 过度使用filter cache

**现象：**

搜索效率低下

Hotthread监控热点搜索的metric。发现在datetime类型属性和geohash的类型属性上使用很多的filtercache。

实际上datetime和geohash属性并不适合用filtercache，因为命中率低下或者更新频繁的原因，filtercache会频繁的释放重新缓存，导致过多的cpu消耗。

**解决方案：**

在更新频繁的字段上，时间，geohash，geoshape字段上不用做filtercache。

### 聚合(aggregations)引发的问题

**现象：**

搜索效率低下

Hotthread监控热点搜索的metric。发现很多聚合搜索。

聚合搜索本身十分消耗性能，它实际是在搜索过程中，聚合你需要的字段。这个过程会使es的collector多使用一个aggregations collector。当聚合的字段有不同的聚合条件时，aggregations collector也会变多。这就降低了搜索的效率。

**解决方案：**

不要实时聚合数据，可以把聚合结果缓存起来，定时更新聚合数据。

# 实用工具

Sql语言集成客户端。

使用T-sql语言进行es搜索操作。在升级中……