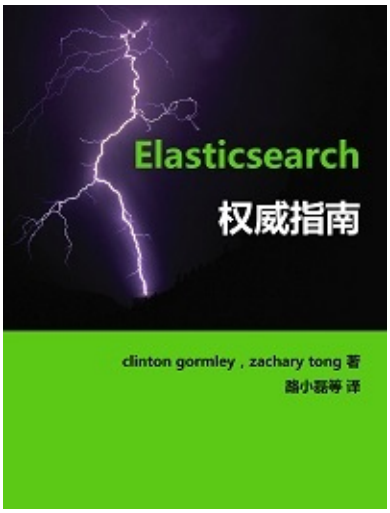
Elasticsearch

权威指南

clinton gormley，zachary tong 著

路小磊等 译

Table of Contents



Elasticsearch 权威指南（中文版）

阅读地址：Elasticsearch权威指南（中文版）

原书地址：Elasticsearch the definitive guide

原作者： clinton gormley，zachary tong

译者： Looly

参与翻译：

- @iridiumcao
- @cvvnx1
- @conan007ai
- @sailxjx
- @wxlfight
- @xieyunzi
- @xdream86
- @conan007ai
- @williamzhao
- @dingusxp
- @birdroidcn

感谢参与翻译的小伙伴们~~

邮箱： loolly@gmail.com

微博： @路小磊

项目地址：

https://github.com/looly/elasticsearch-definitive-guide-cn

http://git.oschina.net/loolly/elasticsearch-definitive-guide-cn

阅读地址：

http://es.xiaoleilu.com/

http://wiki.jikexueyuan.com/project/elasticsearch-definitive-guide-cn/

说明

之前接触Elasticsearch只是最简单的使用，想要深入了解内部功能，借助翻译同时系统学习。由于英语比较菜，第一次翻译文档，如有不妥，欢迎提issue:

github

git@osc

翻译关键字约定

- index -> 索引
- type -> 类型
- token -> 表征
- filter -> 过滤器
- analyser -> 分析器

Pull Request流程

开始我对Pull Request流程不熟悉，后来参考了@numbbbbb的《The Swift Programming Language》协作流程，在此感谢。

1. 首先fork我的项目
2. 把fork过去的项目也就是你的项目clone到你的本地
3. 运行 `git remote add looly git@github.com:looly/elasticsearch-definitive-guide-cn.git` 把我的库添加为远端库
4. 运行 `git pull looly master` 拉取并合并到本地
5. 翻译内容
6. commit后push到自己的库（`git push origin master`）
7. 登录Github在你首页可以看到一个 `pull request` 按钮，点击它，填写一些说明信息，然后提交即可。

1~3是初始化操作，执行一次即可。在翻译前必须执行第4步同步我的库（这样避免冲突），然后执行5~7既可。

注意

1. 文档还未翻译完成，使用gitbook格式，已经翻译完成的章节会陆续提交到gitbook。
2. 为了便于翻译，未翻译部分拷贝自官方英文文档。

入门

Elasticsearch是一个实时分布式搜索和分析引擎。它让你以前所未有的速度处理大数据成为可能。

它用于全文搜索、结构化搜索、分析以及将这三者混合使用：

- 维基百科使用Elasticsearch提供全文搜索并高亮关键字，以及输入实时搜索**(search-as-you-type)**和搜索纠错**(did-you-mean)**等搜索建议功能。
- 英国卫报使用Elasticsearch结合用户日志和社交网络数据提供给他们的编辑以实时的反馈，以便及时了解公众对新发表的文章的回应。
- StackOverflow结合全文搜索与地理位置查询，以及**more-like-this**功能来找到相关的问题和答案。
- Github使用Elasticsearch检索1300亿行的代码。

但是Elasticsearch不仅用于大型企业，它还让像DataDog以及Klout这样的创业公司将最初的想法变成可扩展的解决方案。Elasticsearch可以在你的笔记本上运行，也可以在数以百计的服务器上处理PB级别的数据。

Elasticsearch所涉及到的每一项技术都不是创新或者革命性的，全文搜索，分析系统以及分布式数据库这些早就已经存在了。它的革命性在于将这些独立且有用的技术整合成一个一体化的、实时的应用。它对新用户的门槛很低，当然它也会跟上你技能和需求增长的步伐。

如果你打算看这本书，说明你已经有数据了，但光有数据是不够的，除非你能对这些数据做些什么事情。

很不幸，现在大部分数据库在提取可用知识方面显得异常无能。的确，它们能够通过时间戳或者精确匹配做过滤，但是它们能够进行全文搜索，处理同义词和根据相关性给文档打分吗？它们能根据同一份数据生成分析和聚合的结果吗？最重要的是，它们在没有大量工作进程（线程）的情况下能做到对数据的实时处理吗？

这就是Elasticsearch存在的理由：Elasticsearch鼓励你浏览并利用你的数据，而不是让它烂在数据库里，因为在数据库里实在太难查询了。

Elasticsearch是你新认识的最好的朋友。

为了搜索，你懂的

Elasticsearch是一个基于Apache Lucene(TM)的开源搜索引擎。无论在开源还是专有领域，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。想要使用它，你必须使用Java来作为开发语言并将其直接集成到你的应用中，更糟糕的是，Lucene非常复杂，你需要深入了解检索的相关知识来理解它是如何工作的。

Elasticsearch也使用Java开发并使用Lucene作为其核心来实现所有索引和搜索的功能，但是它的目的是通过简单的`RESTful API`来隐藏Lucene的复杂性，从而让全文搜索变得简单。

不过，Elasticsearch不仅仅是Lucene和全文搜索，我们还能这样去描述它：

- 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

而且，所有的这些功能被集成到一个服务里面，你的应用可以通过简单的`RESTful API`、各种语言的客户端甚至命令行与之交互。

上手Elasticsearch非常容易。它提供了许多合理的缺省值，并对初学者隐藏了复杂的搜索引擎理论。它开箱即用（安装即可使用），只需很少的学习既可在生产环境中使用。

Elasticsearch在Apache 2 license下许可使用，可以免费下载、使用和修改。

随着你对Elasticsearch的理解加深，你可以根据不同的问题领域定制Elasticsearch的高级特性，这一切都是可配置的，并且配置非常灵活。

模糊的历史

多年前，一个叫做Shay Banon的刚结婚不久的失业开发者，由于妻子要去伦敦学习厨师，他便跟着也去了。在他找工作的过程中，为了给妻子构建一个食谱的搜索引擎，他开始构建一个早期版本的Lucene。

直接基于Lucene工作会比较困难，所以Shay开始抽象Lucene代码以便Java程序员可以在应用中添加搜索功能。他发布了他的第一个开源项目，叫做“Compass”。

后来Shay找到一份工作，这份工作处在高性能和内存数据网格的分布式环境中，因此高性能的、实时的、分布式的搜索引擎也是理所当然需要的。然后他决定重写Compass库使其成为一个独立的服务叫做Elasticsearch。

第一个公开版本出现在2010年2月，在那之后Elasticsearch已经成为Github上最受欢迎的项目之一，代码贡献者超过300人。一家主营Elasticsearch的公司就此成立，他们一边提供商业支持一边开发新功能，不过Elasticsearch将永远开源且对所有人可用。

Shay的妻子依旧等待着她的食谱搜索……

安装Elasticsearch

理解Elasticsearch最好的方式是去运行它，让我们开始吧！

安装Elasticsearch唯一的要求是安装官方新版的Java，地址：www.java.com

你可以从 elasticsearch.org/download 下载最新版本的Elasticsearch。

```
curl -L -O http://download.elasticsearch.org/PATH/TO/VERSION.zip <1>
unzip elasticsearch-$VERSION.zip
cd  elasticsearch-$VERSION
```

1. 从 elasticsearch.org/download 获得最新可用的版本号并填入URL中

> 提示：
>
> 在生产环境安装时，除了以上方法，你还可以使用Debian或者RPM安装包，地址在这里：downloads page，或者也可以使用官方提供的 Puppet module 或者 Chef cookbook。

安装Marvel

Marvel是Elasticsearch的管理和监控工具，在开发环境下免费使用。它包含了一个叫做 `Sense` 的交互式控制台，使用户方便的通过浏览器直接与Elasticsearch进行交互。

Elasticsearch线上文档中的很多示例代码都附带一个 `View in Sense` 的链接。点击进去，就会在 `Sense` 控制台打开相应的实例。安装Marvel不是必须的，但是它可以通过在你本地Elasticsearch集群中运行示例代码而增加与此书的互动性。

Marvel是一个插件，可在Elasticsearch目录中运行以下命令来下载和安装：

```
./bin/plugin -i elasticsearch/marvel/latest
```

你可能想要禁用监控，你可以通过以下命令关闭Marvel：

```
echo 'marvel.agent.enabled: false' >> ./config/elasticsearch.yml
```

运行Elasticsearch

Elasticsearch已经准备就绪，执行以下命令可在前台启动：

```
./bin/elasticsearch
```

如果想在后台以守护进程模式运行，添加 `-d` 参数。

打开另一个终端进行测试：

```
curl 'http://localhost:9200/?pretty'
```

你能看到以下返回信息：

```
{
   "status": 200,
   "name": "Shrunken Bones",
   "version": {
      "number": "1.4.0",
      "lucene_version": "4.10"
   },
   "tagline": "You Know, for Search"
}
```

这说明你的ELasticsearch集群已经启动并且正常运行，接下来我们可以开始各种实验了。

集群和节点

节点**(node)**是一个运行着的Elasticsearch实例。集群**(cluster)**是一组具有相同 `cluster.name` 的节点集合，他们协同工作，共享数据并提供故障转移和扩展功能，当然一个节点也可以组成一个集群。

你最好找一个合适的名字来替代 `cluster.name` 的默认值，比如你自己的名字，这样可以防止一个新启动的节点加入到相同网络中的另一个同名的集群中。

你可以通过修改 `config/` 目录下的 `elasticsearch.yml` 文件，然后重启ELasticsearch来做到这一点。当Elasticsearch在前台运行，可以使用 `Ctrl-C` 快捷键终止，或者你可以调用 `shutdown` API来关闭：

```
curl -XPOST 'http://localhost:9200/_shutdown'
```

查看Marvel和Sense

如果你安装了Marvel（作为管理和监控的工具），就可以在浏览器里通过以下地址访问它：

http://localhost:9200/_plugin/marvel/

你可以在Marvel中通过点击 `dashboards`，在下拉菜单中访问**Sense**开发者控制台，或者直接访问以下地址：

http://localhost:9200/_plugin/marvel/sense/

与Elasticsearch交互

如何与Elasticsearch交互取决于你是否使用Java。

Java API

Elasticsearch为Java用户提供了两种内置客户端：

节点客户端(node client)：

节点客户端以无数据节点(none data node)身份加入集群，换言之，它自己不存储任何数据，但是它知道数据在集群中的具体位置，并且能够直接转发请求到对应的节点上。

传输客户端(Transport client)：

这个更轻量的传输客户端能够发送请求到远程集群。它自己不加入集群，只是简单转发请求给集群中的节点。

两个Java客户端都通过9300端口与集群交互，使用Elasticsearch传输协议(Elasticsearch Transport Protocol)。集群中的节点之间也通过9300端口进行通信。如果此端口未开放，你的节点将不能组成集群。

> **TIP**
>
> Java客户端所在的Elasticsearch版本必须与集群中其他节点一致，否则，它们可能互相无法识别。

关于Java API的更多信息请查看相关章节：Java API

基于HTTP协议，以JSON为数据交互格式的RESTful API

其他所有程序语言都可以使用RESTful API，通过9200端口的与Elasticsearch进行通信，你可以使用你喜欢的WEB客户端，事实上，如你所见，你甚至可以通过`curl`命令与Elasticsearch通信。

> **NOTE**
>
> Elasticsearch官方提供了多种程序语言的客户端——Groovy，Javascript， .NET，PHP，Perl，Python，以及 Ruby——还有很多由社区提供的客户端和插件，所有这些可以在文档中找到。

向Elasticsearch发出的请求的组成部分与其它普通的HTTP请求是一样的：

```
curl -X<VERB> '<PROTOCOL>://<HOST>/<PATH>?<QUERY_STRING>' -d '<BODY>'
```

- VERB HTTP方法：`GET`，`POST`，`PUT`，`HEAD`，`DELETE`
- PROTOCOL http或者https协议（只有在Elasticsearch前面有https代理的时候可用）
- HOST Elasticsearch集群中的任何一个节点的主机名，如果是在本地的节点，那么就叫localhost
- PORT Elasticsearch HTTP服务所在的端口，默认为9200
- QUERY_STRING 一些可选的查询请求参数，例如`?pretty`参数将使请求返回更加美观易读的JSON数据
- BODY 一个JSON格式的请求主体（如果请求需要的话）

举例说明，为了计算集群中的文档数量，我们可以这样做：

```
curl -XGET 'http://localhost:9200/_count?pretty' -d '
{
    "query": {
        "match_all": {}
    }
}
```

```
'
```

Elasticsearch返回一个类似 `200 OK` 的HTTP状态码和JSON格式的响应主体（除了 `HEAD` 请求）。上面的请求会得到如下的JSON格式的响应主体：

```
{
    "count" : 0,
    "_shards" : {
        "total" : 5,
        "successful" : 5,
        "failed" : 0
    }
}
```

我们看不到HTTP头是因为我们没有让 `curl` 显示它们，如果要显示，使用 `curl` 命令后跟 `-i` 参数:

```
curl -i -XGET 'localhost:9200/'
```

对于本书的其余部分，我们将简写 `curl` 请求中重复的部分，例如主机名和端口，还有 `curl` 命令本身。

一个完整的请求形如：

```
curl -XGET 'localhost:9200/_count?pretty' -d '
{
    "query": {
        "match_all": {}
    }
}'
```

我们将简写成这样：

```
GET /_count
{
    "query": {
        "match_all": {}
    }
}
```

事实上，在Sense控制台中也使用了与上面相同的格式。

面向文档

应用中的对象很少只是简单的键值列表，更多时候它拥有复杂的数据结构，比如包含日期、地理位置、另一个对象或者数组。

总有一天你会想到把这些对象存储到数据库中。将这些数据保存到由行和列组成的关系数据库中，就好像是把一个丰富，信息表现力强的对象拆散了放入一个非常大的表格中：你不得不拆散对象以适应表模式（通常一列表示一个字段），然后又不得不在查询的时候重建它们。

Elasticsearch是面向文档**(document oriented)**的，这意味着它可以存储整个对象或文档**(document)**。然而它不仅仅是存储，还会索引**(index)**每个文档的内容使之可以被搜索。在Elasticsearch中，你可以对文档（而非成行成列的数据）进行索引、搜索、排序、过滤。这种理解数据的方式与以往完全不同，这也是Elasticsearch能够执行复杂的全文搜索的原因之一。

JSON

ELasticsearch使用**Javascript**对象符号**(JavaScript Object Notation)**，也就是**JSON**，作为文档序列化格式。JSON现在已经被大多语言所支持，而且已经成为NoSQL领域的标准格式。它简洁、简单且容易阅读。

以下使用JSON文档来表示一个用户对象：

```
{
    "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"
}
```

尽管原始的`user`对象很复杂，但它的结构和对象的含义已经被完整的体现在JSON中了，在Elasticsearch中将对象转化为JSON并做索引要比在表结构中做相同的事情简单的多。

> **NOTE**
>
> 尽管几乎所有的语言都有相应的模块用于将任意数据结构转换为JSON，但每种语言处理细节不同。具体请查看“`serialization`” or “`marshalling`”两个用于处理JSON的模块。Elasticsearch官方客户端会自动为你序列化和反序列化JSON。

开始第一步

我们现在开始进行一个简单教程，它涵盖了一些基本的概念介绍，比如索引**(indexing)**、搜索**(search)**以及聚合**(aggregations)**。通过这个教程，我们可以让你对Elasticsearch能做的事以及其易用程度有一个大致的感觉。

我们接下来将陆续介绍一些术语和基本的概念，但就算你没有马上完全理解也没有关系。我们将在本书的各个章节中更加深入的探讨这些内容。

所以，坐下来，开始以旋风般的速度来感受Elasticsearch的能力吧！

让我们建立一个员工目录

假设我们刚好在**Megacorp**工作，这时人力资源部门出于某种目的需要让我们创建一个员工目录，这个目录用于促进人文关怀和用于实时协同工作，所以它有以下不同的需求：

- 数据能够包含多个值的标签、数字和纯文本。
- 检索任何员工的所有信息。
- 支持结构化搜索，例如查找30岁以上的员工。
- 支持简单的全文搜索和更复杂的短语**(phrase)**搜索
- 高亮搜索结果中的关键字
- 能够利用图表管理分析这些数据

索引员工文档

我们首先要做的是存储员工数据，每个文档代表一个员工。在Elasticsearch中存储数据的行为就叫做索引**(indexing)**，不过在索引之前，我们需要明确数据应该存储在哪里。

在Elasticsearch中，文档归属于一种类型**(type)**,而这些类型存在于索引**(index)**中，我们可以画一些简单的对比图来类比传统关系型数据库：

```
Relational DB -> Databases -> Tables -> Rows -> Columns
Elasticsearch -> Indices   -> Types  -> Documents -> Fields
```

Elasticsearch集群可以包含多个索引**(indices)**（数据库），每一个索引可以包含多个类型**(types)**（表），每一个类型包含多个文档**(documents)**（行），然后每个文档包含多个字段**(Fields)**（列）。

> ## 「索引」含义的区分
>
> 你可能已经注意到索引**(index)**这个词在Elasticsearch中有着不同的含义，所以有必要在此做一下区分:
>
> - 索引（名词） 如上文所述，一个索引**(index)**就像是传统关系数据库中的数据库，它是相关文档存储的地方，index的复数是**indices** 或**indexes**。
> - 索引（动词） 「索引一个文档」表示把一个文档存储到索引（名词）里，以便它可以被检索或者查询。这很像SQL中的`INSERT`关键字，差别是，如果文档已经存在，新的文档将覆盖旧的文档。
> - 倒排索引 传统数据库为特定列增加一个索引，例如B-Tree索引来加速检索。Elasticsearch和Lucene使用一种叫做倒排索引**(inverted index)**的数据结构来达到相同目的。

默认情况下，文档中的所有字段都会被索引（拥有一个倒排索引），只有这样他们才是可被搜索的。

我们将会在倒排索引章节中更详细的讨论。

所以为了创建员工目录，我们将进行如下操作：

- 为每个员工的文档**(document)**建立索引，每个文档包含了相应员工的所有信息。
- 每个文档的类型为 `employee` 。
- `employee` 类型归属于索引 `megacorp` 。
- `megacorp` 索引存储在Elasticsearch集群中。

实际上这些都是很容易的（尽管看起来有许多步骤）。我们能通过一个命令执行完成的操作：

```
PUT /megacorp/employee/1
{
    "first_name" : "John",
    "last_name" :  "Smith",
    "age" :        25,
    "about" :      "I love to go rock climbing",
    "interests": [ "sports", "music" ]
}
```

我们看到path: `/megacorp/employee/1` 包含三部分信息：

名字	说明
megacorp	索引名
employee	类型名
1	这个员工的ID

请求实体（JSON文档），包含了这个员工的所有信息。他的名字叫“John Smith”，25岁，喜欢攀岩。

很简单吧！它不需要你做额外的管理操作，比如创建索引或者定义每个字段的数据类型。我们能够直接索引文档，Elasticsearch已经内置所有的缺省设置，所有管理操作都是透明的。

接下来，让我们在目录中加入更多员工信息：

```
PUT /megacorp/employee/2
{
    "first_name" :  "Jane",
    "last_name" :   "Smith",
    "age" :         32,
    "about" :       "I like to collect rock albums",
    "interests":  [ "music" ]
}

PUT /megacorp/employee/3
{
    "first_name" :  "Douglas",
    "last_name" :   "Fir",
    "age" :         35,
    "about":        "I like to build cabinets",
    "interests":  [ "forestry" ]
}
```

检索文档

现在Elasticsearch中已经存储了一些数据，我们可以根据业务需求开始工作了。第一个需求是能够检索单个员工的信息。

这对于Elasticsearch来说非常简单。我们只要执行HTTP GET请求并指出文档的“地址”——索引、类型和ID既可。根据这三部分信息，我们就可以返回原始JSON文档：

```
GET /megacorp/employee/1
```

响应的内容中包含一些文档的元信息，John Smith的原始JSON文档包含在 `_source` 字段中。

```
{
  "_index" :   "megacorp",
  "_type" :    "employee",
  "_id" :      "1",
  "_version" : 1,
  "found" :    true,
  "_source" :  {
      "first_name" :  "John",
      "last_name" :   "Smith",
      "age" :         25,
      "about" :       "I love to go rock climbing",
      "interests":  [ "sports", "music" ]
  }
}
```

> 我们通过HTTP方法 `GET` 来检索文档，同样的，我们可以使用 `DELETE` 方法删除文档，使用 `HEAD` 方法检查某文档是否存在。如果想更新已存在的文档，我们只需再 `PUT` 一次。

简单搜索

`GET` 请求非常简单——你能轻松获取你想要的文档。让我们来进一步尝试一些东西，比如简单的搜索！

我们尝试一个最简单的搜索全部员工的请求：

```
GET /megacorp/employee/_search
```

你可以看到我们依然使用 `megacorp` 索引和 `employee` 类型，但是我们在结尾使用关键字 `_search` 来取代原来的文档ID。响应内容的 `hits` 数组中包含了我们所有的三个文档。默认情况下搜索会返回前10个结果。

```
{
   "took":      6,
   "timed_out": false,
   "_shards": { ... },
   "hits": {
      "total":      3,
      "max_score":  1,
      "hits": [
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "3",
            "_score":         1,
            "_source": {
               "first_name":  "Douglas",
               "last_name":   "Fir",
               "age":         35,
               "about":       "I like to build cabinets",
               "interests": [ "forestry" ]
            }
         },
```

```
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "1",
            "_score":         1,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "2",
            "_score":         1,
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

> 注意：
>
> 响应内容不仅会告诉我们哪些文档被匹配到，而且这些文档内容完整的被包含在其中—我们在给用户展示搜索结果时需要用到的所有信息都有了。

接下来，让我们搜索姓氏中包含**“Smith”**的员工。要做到这一点，我们将在命令行中使用轻量级的搜索方法。这种方法常被称作查询字符串**(query string)**搜索，因为我们像传递URL参数一样去传递查询语句：

```
GET /megacorp/employee/_search?q=last_name:Smith
```

我们在请求中依旧使用 `_search` 关键字，然后将查询语句传递给参数 `q=` 。这样就可以得到所有姓氏为Smith的结果：

```
{
   ...
   "hits": {
      "total":      2,
      "max_score":  0.30685282,
      "hits": [
         {
            ...
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            ...
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

使用DSL语句查询

查询字符串搜索便于通过命令行完成特定**(ad hoc)**的搜索，但是它也有局限性（参阅简单搜索章节）。Elasticsearch提供丰富且灵活的查询语言叫做**DSL**查询**(Query DSL)**,它允许你构建更加复杂、强大的查询。

DSL(Domain Specific Language特定领域语言**)**以JSON请求体的形式出现。我们可以这样表示之前关于“Smith”的查询:

```
GET /megacorp/employee/_search
{
    "query" : {
        "match" : {
            "last_name" : "Smith"
        }
    }
}
```

这会返回与之前查询相同的结果。你可以看到有些东西改变了，我们不再使用查询字符串**(query string)**做为参数，而是使用请求体代替。这个请求体使用JSON表示，其中使用了 `match` 语句（查询类型之一，具体我们以后会学到）。

更复杂的搜索

我们让搜索稍微再变的复杂一些。我们依旧想要找到姓氏为“Smith”的员工，但是我们只想得到年龄大于30岁的员工。我们的语句将添加过滤器**(filter)**,它使得我们高效率的执行一个结构化搜索：

```
GET /megacorp/employee/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "range" : {
                    "age" : { "gt" : 30 } <1>
                }
            },
            "query" : {
                "match" : {
                    "last_name" : "smith" <2>
                }
            }
        }
    }
}
```

- <1> 这部分查询属于区间过滤器**(range filter)**,它用于查找所有年龄大于30岁的数据—— `gt` 为"greater than"的缩写。
- <2> 这部分查询与之前的 `match` 语句**(query)**一致。

现在不要担心语法太多，我们将会在以后详细的讨论。你只要知道我们添加了一个过滤器**(filter)**用于执行区间搜索，然后重复利用了之前的 `match` 语句。现在我们的搜索结果只显示了一个32岁且名字是“Jane Smith”的员工：

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.30685282,
      "hits": [
         {
            ...
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
```

```
    }
}
```

全文搜索

到目前为止搜索都很简单：搜索特定的名字，通过年龄筛选。让我们尝试一种更高级的搜索，全文搜索——一种传统数据库很难实现的功能。

我们将会搜索所有喜欢**“rock climbing”**的员工：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match" : {
            "about" : "rock climbing"
        }
    }
}
```

你可以看到我们使用了之前的`match`查询，从`about`字段中搜索**"rock climbing"**，我们得到了两个匹配文档：

```
{
   ...
   "hits": {
      "total":      2,
      "max_score":  0.16273327,
      "hits": [
         {
            ...
            "_score":         0.16273327, <1>
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            ...
            "_score":         0.016878016, <2>
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

- <1><2> 结果相关性评分。

默认情况下，Elasticsearch根据结果相关性评分来对结果集进行排序，所谓的「结果相关性评分」就是文档与查询条件的匹配程度。很显然，排名第一的`John Smith`的`about`字段明确的写到**“rock climbing”**。

但是为什么`Jane Smith`也会出现在结果里呢？原因是**“rock”**在她的`abuot`字段中被提及了。因为只有**“rock”**被提及而**“climbing”**没有，所以她的`_score`要低于John。

这个例子很好的解释了Elasticsearch如何在各种文本字段中进行全文搜索，并且返回相关性最大的结果集。相关性**(relevance)**的概念在Elasticsearch中非常重要，而这个概念在传统关系型数据库中是不可想象的，因为传统数据库对记录的查询只有匹配或者不匹配。

短语搜索

目前我们可以在字段中搜索单独的一个词，这挺好的，但是有时候你想要确切的匹配若干个单词或者短语**(phrases)**。例如我们想要查询同时包含"rock"和"climbing"（并且是相邻的）的员工记录。

要做到这个，我们只要将 `match` 查询变更为 `match_phrase` 查询即可:

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
    }
}
```

毫无疑问，该查询返回John Smith的文档：

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.23013961,
      "hits": [
         {
            ...
            "_score":         0.23013961,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         }
      ]
   }
}
```

高亮我们的搜索

很多应用喜欢从每个搜索结果中高亮**(highlight)**匹配到的关键字，这样用户可以知道为什么这些文档和查询相匹配。在Elasticsearch中高亮片段是非常容易的。

让我们在之前的语句上增加 `highlight` 参数：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
    },
    "highlight": {
        "fields" : {
            "about" : {}
        }
    }
}
```

当我们运行这个语句时，会命中与之前相同的结果，但是在返回结果中会有一个新的部分叫做 `highlight`，这里包含了来自 `about` 字段中的文本，并且用 `<em></em>` 来标识匹配到的单词。

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.23013961,
      "hits": [
         {
            ...
            "_score":         0.23013961,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            },
            "highlight": {
               "about": [
                  "I love to go <em>rock</em> <em>climbing</em>" <1>
               ]
            }
         }
      ]
   }
}
```

- <1> 原有文本中高亮的片段

你可以在高亮章节阅读更多关于搜索高亮的部分。

分析

最后，我们还有一个需求需要完成：允许管理者在职员目录中进行一些分析。 Elasticsearch有一个功能叫做聚合**(aggregations)**，它允许你在数据上生成复杂的分析统计。它很像SQL中的`GROUP BY`但是功能更强大。

举个例子，让我们找到所有职员中最大的共同点（兴趣爱好）是什么：

```
GET /megacorp/employee/_search
{
  "aggs": {
    "all_interests": {
      "terms": { "field": "interests" }
    }
  }
}
```

暂时先忽略语法只看查询结果：

```
{
   ...
   "hits": { ... },
   "aggregations": {
      "all_interests": {
         "buckets": [
            {
               "key":       "music",
               "doc_count": 2
            },
            {
               "key":       "forestry",
               "doc_count": 1
            },
            {
               "key":       "sports",
               "doc_count": 1
            }
         ]
      }
   }
}
```

我们可以看到两个职员对音乐有兴趣，一个喜欢林学，一个喜欢运动。这些数据并没有被预先计算好，它们是实时的从匹配查询语句的文档中动态计算生成的。如果我们想知道所有姓"Smith"的人最大的共同点（兴趣爱好），我们只需要增加合适的语句既可：

```
GET /megacorp/employee/_search
{
  "query": {
    "match": {
      "last_name": "smith"
    }
  },
  "aggs": {
    "all_interests": {
      "terms": {
        "field": "interests"
      }
    }
  }
}
```

`all_interests`聚合已经变成只包含和查询语句相匹配的文档了：

```
   ...
```

```
  "all_interests": {
     "buckets": [
        {
           "key": "music",
           "doc_count": 2
        },
        {
           "key": "sports",
           "doc_count": 1
        }
     ]
  }
```

聚合也允许分级汇总。例如，让我们统计每种兴趣下职员的平均年龄：

```
GET /megacorp/employee/_search
{
    "aggs" : {
        "all_interests" : {
            "terms" : { "field" : "interests" },
            "aggs" : {
                "avg_age" : {
                    "avg" : { "field" : "age" }
                }
            }
        }
    }
}
```

虽然这次返回的聚合结果有些复杂，但任然很容易理解：

```
  ...
  "all_interests": {
     "buckets": [
        {
           "key": "music",
           "doc_count": 2,
           "avg_age": {
              "value": 28.5
           }
        },
        {
           "key": "forestry",
           "doc_count": 1,
           "avg_age": {
              "value": 35
           }
        },
        {
           "key": "sports",
           "doc_count": 1,
           "avg_age": {
              "value": 25
           }
        }
     ]
  }
```

该聚合结果比之前的聚合结果要更加丰富。我们依然得到了兴趣以及数量（指具有该兴趣的员工人数）的列表，但是现在每个兴趣额外拥有 `avg_age` 字段来显示具有该兴趣员工的平均年龄。

即使你还不理解语法，但你也可以大概感觉到通过这个特性可以完成相当复杂的聚合工作，你可以处理任何类型的数据。

教程小结

希望这个简短的教程能够很好的描述Elasticsearch的功能。当然这只是一些皮毛，为了保持简短，还有很多的特性未提及——像推荐、定位、渗透、模糊以及部分匹配等。但这也突出了构建高级搜索功能是多么的容易。无需配置，只需要添加数据然后开始搜索！

可能有些语法让你觉得有些困惑，或者在微调方面有些疑问。那么，本书的其余部分将深入这些问题的细节，让你全面了解Elasticsearch的工作过程。

分布式的特性

在章节的开始我们提到Elasticsearch可以扩展到上百（甚至上千）的服务器来处理PB级的数据。然而我们的教程只是给出了一些使用Elasticsearch的例子，并未涉及相关机制。Elasticsearch为分布式而生，而且它的设计隐藏了分布式本身的复杂性。

Elasticsearch在分布式概念上做了很大程度上的透明化，在教程中你不需要知道任何关于分布式系统、分片、集群发现或者其他大量的分布式概念。所有的教程你即可以运行在你的笔记本上，也可以运行在拥有100个节点的集群上，其工作方式是一样的。

Elasticsearch致力于隐藏分布式系统的复杂性。以下这些操作都是在底层自动完成的：

- 将你的文档分区到不同的容器或者分片**(shards)**中，它们可以存在于一个或多个节点中。
- 将分片均匀的分配到各个节点，对索引和搜索做负载均衡。
- 冗余每一个分片，防止硬件故障造成的数据丢失。
- 将集群中任意一个节点上的请求路由到相应数据所在的节点。
- 无论是增加节点，还是移除节点，分片都可以做到无缝的扩展和迁移。

当你阅读本书时，你可以遇到关于Elasticsearch分布式特性的补充章节。这些章节将教给你如何扩展集群和故障转移，如何处理文档存储，如何执行分布式搜索，分片是什么以及如何工作。

这些章节不是必读的——不懂这些内部机制也可以使用Elasticsearch的。但是这些能够帮助你更深入和完整的了解Elasticsearch。你可以略读它们，然后在你需要更深入的理解时再回头翻阅。

下一步

现在你对Elasticsearch可以做些什么以及其易用程度有了大概的了解。Elasticsearch致力于降低学习成本和轻松配置。学习Elasticsearch最好的方式就是开始使用它：开始索引和检索吧！

当然，你越是了解Elasticsearch，你的生产力就越高。你越是详细告诉Elasticsearch你的应用的数据特点，你就越能得到准确的输出。

本书其余部分将帮助你从新手晋级到专家。每一个章节都会阐述一个要点，并且会包含专家级别的技巧。如果你只是刚起步，那么这些技巧可能暂时和你无关。Elasticsearch有合理的默认配置而且可以在没有用户干预的情况下做正确的事情。当需要提升性能时你可以随时回顾这些章节。

集群内部工作方式

> 补充章节
>
> 正如之前提及的，这是关于Elasticsearch在分布式环境下工作机制的一些补充章节的第一部分。这个章节我们解释一些通用的术语，例如集群**(cluster)**、节点**(node)**和分片**(shard)**，Elasticsearch的扩展机制，以及它如何处理硬件故障。
>
> 尽管这章不是必读的——你在使用Elasticsearch的时候可以长时间甚至永远都不必担心分片、复制和故障转移——但是它会帮助你理解Elasticsearch内部的工作流程，你可以先跳过这章，以后再来查阅。

Elasticsearch用于构建高可用和可扩展的系统。扩展的方式可以是购买更好的服务器(纵向扩展**(vertical scale or scaling up)**)或者购买更多的服务器（横向扩展**(horizontal scale or scaling out)**）。

Elasticsearch虽然能从更强大的硬件中获得更好的性能，但是纵向扩展有它的局限性。真正的扩展应该是横向的，它通过增加节点来均摊负载和增加可靠性。

对于大多数数据库而言，横向扩展意味着你的程序将做非常大的改动才能利用这些新添加的设备。对比来说，Elasticsearch天生就是分布式的：它知道如何管理节点来提供高扩展和高可用。这意味着你的程序不需要关心这些。

在这章我们将探索如何创建你的集群**(cluster)**、节点**(node)**和分片**(shards)**，使其按照你的需求进行扩展，并保证在硬件故障时数据依旧安全。

空集群

如果我们启动一个单独的节点，它还没有数据和索引，这个集群看起来就像图1。

图1：只有一个空节点的集群

一个节点**(node)**就是一个Elasticsearch实例，而一个集群**(cluster)**由一个或多个节点组成，它们具有相同的`cluster.name`，它们协同工作，分享数据和负载。当加入新的节点或者删除一个节点时，集群就会感知到并平衡数据。

集群中一个节点会被选举为主节点**(master)**,它将临时管理集群级别的一些变更，例如新建或删除索引、增加或移除节点等。主节点不参与文档级别的变更或搜索，这意味着在流量增长的时候，该主节点不会成为集群的瓶颈。任何节点都可以成为主节点。我们例子中的集群只有一个节点，所以它会充当主节点的角色。

做为用户，我们能够与集群中的任何节点通信，包括主节点。每一个节点都知道文档存在于哪个节点上，它们可以转发请求到相应的节点上。我们访问的节点负责收集各节点返回的数据，最后一起返回给客户端。这一切都由Elasticsearch处理。

集群健康

在Elasticsearch集群中可以监控统计很多信息，但是只有一个是最重要的：集群健康**(cluster health)**。集群健康有三种状态：`green`、`yellow`或`red`。

```
GET /_cluster/health
```

在一个没有索引的空集群中运行如上查询，将返回这些信息：

```
{
   "cluster_name":          "elasticsearch",
   "status":                "green", <1>
   "timed_out":             false,
   "number_of_nodes":       1,
   "number_of_data_nodes":  1,
   "active_primary_shards": 0,
   "active_shards":         0,
   "relocating_shards":     0,
   "initializing_shards":   0,
   "unassigned_shards":     0
}
```

- <1> `status`是我们最感兴趣的字段

`status`字段提供一个综合的指标来表示集群的的服务状况。三种颜色各自的含义：

颜色	意义
`green`	所有主要分片和复制分片都可用
`yellow`	所有主要分片可用，但不是所有复制分片都可用
`red`	不是所有的主要分片都可用

在接下来的章节，我们将说明什么是主要分片**(primary shard)**和复制分片**(replica shard)**，并说明这些颜色（状态）在实际环境中的意义。

添加索引

为了将数据添加到Elasticsearch，我们需要索引**(index)**——一个存储关联数据的地方。实际上，索引只是一个用来指向一个或多个分片**(shards)**的**“**逻辑命名空间**(logical namespace)”**.

一个分片**(shard)**是一个最小级别**“**工作单元**(worker unit)”**,它只是保存了索引中所有数据的一部分。在接下来的《深入分片》一章，我们将详细说明分片的工作原理，但是现在我们只要知道分片就是一个Lucene实例，并且它本身就是一个完整的搜索引擎。我们的文档存储在分片中，并且在分片中被索引，但是我们的应用程序不会直接与它们通信，取而代之的是，直接与索引通信。

分片是Elasticsearch在集群中分发数据的关键。把分片想象成数据的容器。文档存储在分片中，然后分片分配到你集群中的节点上。当你的集群扩容或缩小，Elasticsearch将会自动在你的节点间迁移分片，以使集群保持平衡。

分片可以是主分片**(primary shard)**或者是复制分片**(replica shard)**。你索引中的每个文档属于一个单独的主分片，所以主分片的数量决定了索引最多能存储多少数据。

> 理论上主分片能存储的数据大小是没有限制的，限制取决于你实际的使用情况。分片的最大容量完全取决于你的使用状况：硬件存储的大小、文档的大小和复杂度、如何索引和查询你的文档，以及你期望的响应时间。

复制分片只是主分片的一个副本，它可以防止硬件故障导致的数据丢失，同时可以提供读请求，比如搜索或者从别的shard取回文档。

当索引创建完成的时候，主分片的数量就固定了，但是复制分片的数量可以随时调整。

让我们在集群中唯一一个空节点上创建一个叫做 `blogs` 的索引。默认情况下，一个索引被分配5个主分片，但是为了演示的目的，我们只分配3个主分片和一个复制分片（每个主分片都有一个复制分片）：

```
PUT /blogs
{
   "settings" : {
      "number_of_shards" : 3,
      "number_of_replicas" : 1
   }
}
```

附带索引的单一节点集群：

我们的集群现在看起来就像上图——三个主分片都被分配到 `Node 1` 。如果我们现在检查集群健康**(cluster-health)**，我们将见到以下信息：

```
{
   "cluster_name":          "elasticsearch",
   "status":                "yellow", <1>
   "timed_out":             false,
   "number_of_nodes":       1,
   "number_of_data_nodes":  1,
   "active_primary_shards": 3,
   "active_shards":         3,
```

```
    "relocating_shards":     0,
    "initializing_shards":   0,
    "unassigned_shards":     3 <2>
}
```

- <1> 集群的状态现在是 `yellow`
- <2> 我们的三个复制分片还没有被分配到节点上

集群的健康状态 `yellow` 表示所有的主分片**(primary shards)**启动并且正常运行了——集群已经可以正常处理任何请求——但是复制分片**(replica shards)**还没有全部可用。事实上所有的三个复制分片现在都是 `unassigned` 状态——它们还未被分配给节点。在同一个节点上保存相同的数据副本是没有必要的，如果这个节点故障了，那所有的数据副本也会丢失。

现在我们的集群已经功能完备，但是依旧存在因硬件故障而导致数据丢失的风险。

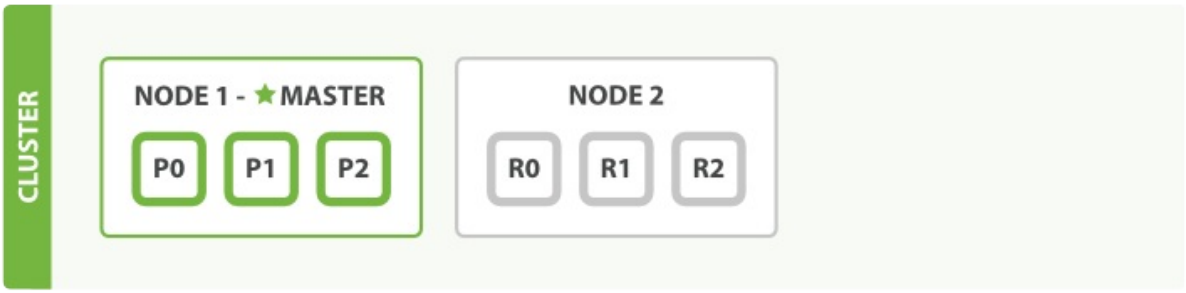
增加故障转移

在单一节点上运行意味着有单点故障的风险——没有数据备份。幸运的是，要防止单点故障，我们唯一需要做的就是启动另一个节点。

> ### 启动第二个节点
>
> 为了测试在增加第二个节点后发生了什么，你可以使用与第一个节点相同的方式启动第二个节点（《运行Elasticsearch》一章），而且命令行在同一个目录——一个节点可以启动多个Elasticsearch实例。
>
> 只要第二个节点与第一个节点有相同的 `cluster.name` （请看 `./config/elasticsearch.yml` 文件），它就能自动发现并加入第一个节点所在的集群。如果没有，检查日志找出哪里出了问题。这可能是网络广播被禁用，或者防火墙阻止了节点通信。

如果我们启动了第二个节点，这个集群看起来就像下图。

双节点集群——所有的主分片和复制分片都已分配:



第二个节点已经加入集群，三个复制分片**(replica shards)**也已经被分配了——分别对应三个主分片，这意味着在丢失任意一个节点的情况下依旧可以保证数据的完整性。

文档的索引将首先被存储在主分片中，然后并发复制到对应的复制节点上。这可以确保我们的数据在主节点和复制节点上都可以被检索。

`cluster-health` 现在的状态是 `green`，这意味着所有的6个分片（三个主分片和三个复制分片）都已可用：

```
{
   "cluster_name":          "elasticsearch",
   "status":                "green", <1>
   "timed_out":             false,
   "number_of_nodes":       2,
   "number_of_data_nodes":  2,
   "active_primary_shards": 3,
   "active_shards":         6,
   "relocating_shards":     0,
   "initializing_shards":   0,
   "unassigned_shards":     0
}
```

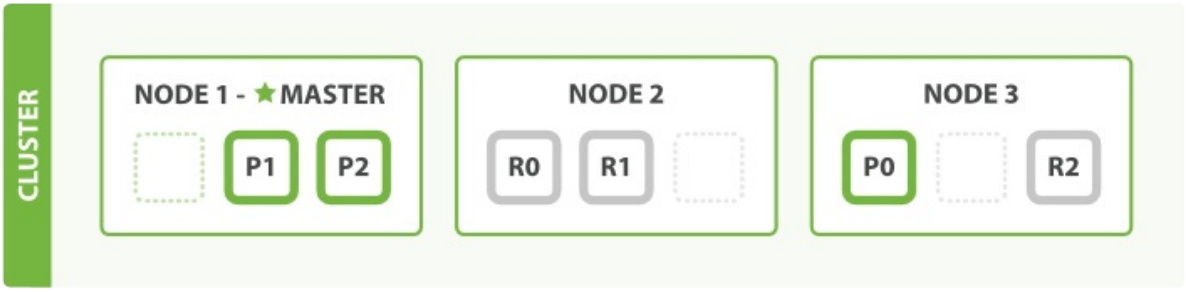
- <1> 集群的状态是 `green` .

我们的集群不仅是功能完备的，而且是高可用的。

横向扩展

随着应用需求的增长，我们该如何扩展？如果我们启动第三个节点，我们的集群会重新组织自己，就像图4：

图4：包含3个节点的集群——分片已经被重新分配以平衡负载：



`Node3` 包含了分别来自 `Node 1` 和 `Node 2` 的一个分片，这样每个节点就有两个分片，和之前相比少了一个，这意味着每个节点上的分片将获得更多的硬件资源（CPU、RAM、I/O）。

分片本身就是一个完整的搜索引擎，它可以使用单一节点的所有资源。我们拥有6个分片（3个主分片和三个复制分片），最多可以扩展到6个节点，每个节点上有一个分片，每个分片可以100%使用这个节点的资源。

继续扩展

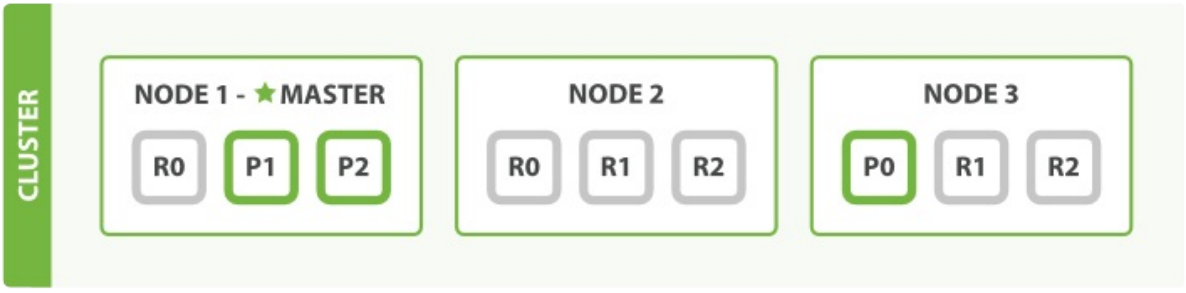
如果我们要扩展到6个以上的节点，要怎么做？

主分片的数量在创建索引时已经确定。实际上，这个数量定义了能存储到索引里数据的最大数量（实际的数量取决于你的数据、硬件和应用场景）。然而，主分片或者复制分片都可以处理读请求——搜索或文档检索，所以数据的冗余越多，我们能处理的搜索吞吐量就越大。

复制分片的数量可以在运行中的集群中动态地变更，这允许我们可以根据需求扩大或者缩小规模。让我们把复制分片的数量从原来的 `1` 增加到 `2` ：

```
PUT /blogs/_settings
{
   "number_of_replicas" : 2
}
```

图5：增加 `number_of_replicas` 到2：



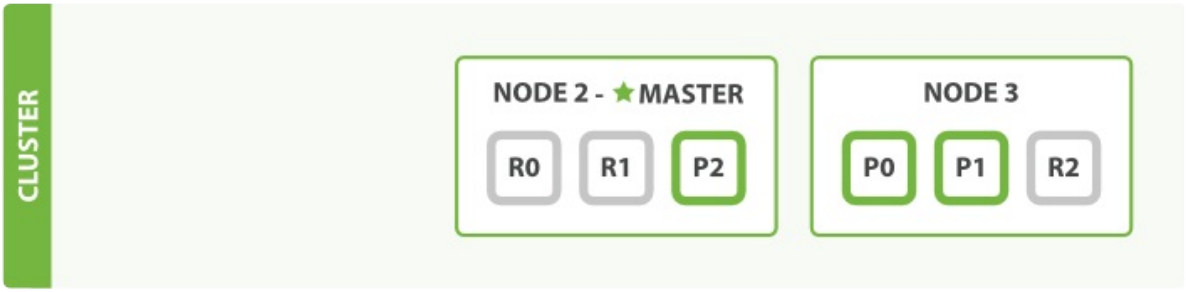
从图中可以看出， `blogs` 索引现在有9个分片：3个主分片和6个复制分片。这意味着我们能够扩展到9个节点，再次变成每个节点一个分片。这样使我们的搜索性能相比原始的三节点集群增加三倍。

> 当然，在同样数量的节点上增加更多的复制分片并不能提高性能，因为这样做的话平均每个分片的所占有的硬件资源就减少了（译者注：大部分请求都聚集到了分片少的节点，导致一个节点吞吐量太大，反而降低性能），你需要增加硬件来提高吞吐量。
>
> 不过这些额外的复制节点使我们有更多的冗余：通过以上对节点的设置，我们能够承受两个节点故障而不丢失数据。

应对故障

我们已经说过Elasticsearch可以应对节点失效，所以让我们继续尝试。如果我们杀掉第一个节点的进程（以下简称杀掉节点），我们的集群看起来就像这样：

图5：杀掉第一个节点后的集群



我们杀掉的节点是一个主节点。一个集群必须要有一个主节点才能使其功能正常，所以集群做的第一件事就是各节点选举了一个新的主节点：`Node 2`。

主分片`1`和`2`在我们杀掉`Node 1`时已经丢失，我们的索引在丢失主分片时不能正常工作。如果此时我们检查集群健康，我们将看到状态`red`：不是所有主节点都可用！

幸运的是丢失的两个主分片的完整拷贝存在于其他节点上，所以新主节点做的第一件事是把这些在`Node 2`和`Node 3`上的复制分片升级为主分片，这时集群健康回到`yellow`状态。这个提升是瞬间完成的，就好像按了一下开关。

为什么集群健康状态是`yellow`而不是`green`？我们有三个主分片，但是我们指定了每个主分片对应两个复制分片，当前却只有一个复制分片被分配，这就是集群状态无法达到`green`的原因，不过不用太担心这个：当我们杀掉`Node 2`，我们的程序依然可以在没有丢失数据的情况下继续运行，因为`Node 3`还有每个分片的拷贝。

如果我们重启`Node 1`，集群将能够重新分配丢失的复制分片，集群状况与上一节的图**5：增加number_of_replicas到2**类似。如果`Node 1`依旧有旧分片的拷贝，它将会尝试再利用它们，它只会从主分片上复制在故障期间有数据变更的那一部分。

现在你应该对分片如何使Elasticsearch可以水平扩展并保证数据安全有了一个清晰的认识。接下来我们将会讨论分片生命周期的更多细节。

数据吞吐

无论程序怎么写，意图是一样的：组织数据为我们的目标所服务。但数据并不只是由随机比特和字节组成，我们在数据节点间建立关联来表示现实世界中的实体或者“某些东西”。属于同一个人的名字和Email地址会有更多的意义。

在现实世界中，并不是所有相同类型的实体看起来都是一样的。一个人可能有一个家庭电话号码，另一个人可能只有一个手机号码，有些人可能两者都有。一个人可能有三个Email地址，其他人可能没有。西班牙人可能有两个姓氏，但是英国人（英语系国家的人）可能只有一个。

面向对象编程语言流行的原因之一，是我们可以用对象来表示和处理现实生活中那些有着潜在关系和复杂结构的实体。到目前为止，这种方式还不错。

但当我们想存储这些实体时问题便来了。传统上，我们以行和列的形式把数据存储在关系型数据库中，相当于使用电子表格。这种固定的存储方式导致对象的灵活性不复存在了。

但是如何能以对象的形式存储对象呢？相对于围绕表格去为我们的程序去建模，我们可以专注于使用数据，把对象本来的灵活性找回来。

对象**(object)**是一种语言相关，记录在内存中的的数据结构。为了在网络间发送，或者存储它，我们需要一些标准的格式来表示它。JSON (JavaScript Object Notation)是一种可读的以文本来表示对象的方式。它已经成为NoSQL世界中数据交换的一种事实标准。当对象被序列化为JSON，它就成为**JSON**文档**(JSON document)**了。

Elasticsearch是一个分布式的文档**(document)**存储引擎。它可以实时存储并检索复杂数据结构——序列化的JSON文档。换言说，一旦文档被存储在Elasticsearch中，它就可以在集群的任一节点上被检索。

当然，我们不仅需要存储数据，还要快速的批量查询。虽然已经有很多NoSQL的解决方案允许我们以文档的形式存储对象，但它们依旧需要考虑如何查询这些数据，以及哪些字段需要被索引以便检索时更加快速。

在Elasticsearch中，每一个字段的数据都是默认被索引的。也就是说，每个字段专门有一个反向索引用于快速检索。而且，与其它数据库不同，它可以在同一个查询中利用所有的这些反向索引，以惊人的速度返回结果。

在这一章我们将探讨如何使用API来创建、检索、更新和删除文档。目前，我们并不关心数据如何在文档中以及如何查询他们。所有我们关心的是文档如何安全在Elasticsearch中存储，以及如何让它们返回。

什么是文档？

程序中大多的实体或对象能够被序列化为包含键值对的JSON对象，键**(key)**是字段**(field)**或属性**(property)**的名字，值**(value)**可以是字符串、数字、布尔类型、另一个对象、值数组或者其他特殊类型，比如表示日期的字符串或者表示地理位置的对象。

```
{
    "name":         "John Smith",
    "age":          42,
    "confirmed":    true,
    "join_date":    "2014-06-01",
    "home": {
        "lat":      51.5,
        "lon":      0.1
    },
    "accounts": [
        {
            "type": "facebook",
            "id":   "johnsmith"
        },
        {
            "type": "twitter",
            "id":   "johnsmith"
        }
    ]
}
```

通常，我们可以认为对象**(object)**和文档**(document)**是等价相通的。不过，他们还是有所差别：对象(Object)是一个JSON结构体——类似于哈希、hashmap、字典或者关联数组；对象(Object)中还可能包含其他对象(Object)。 在Elasticsearch中，文档**(document)**这个术语有着特殊含义。它特指最顶层结构或者根对象**(root object)**序列化成的JSON数据（以唯一ID标识并存储于Elasticsearch中）。

文档元数据

一个文档不只有数据。它还包含了元数据**(metadata)**——关于文档的信息。三个必须的元数据节点是：

节点	说明
`_index`	文档存储的地方
`_type`	文档代表的对象的类
`_id`	文档的唯一标识

`_index`

索引**(index)**类似于关系型数据库里的“数据库”——它是我们存储和索引关联数据的地方。

> 提示：
>
> 事实上，我们的数据被存储和索引在分片**(shards)**中，索引只是一个把一个或多个分片分组在一起的逻辑空间。然而，这只是一些内部细节——我们的程序完全不用关心分片。对于我们的程序而言，文档存储在索引**(index)**中。剩下的细节由Elasticsearch关心既可。

我们将会在《索引管理》章节中探讨如何创建并管理索引，但现在，我们将让Elasticsearch为我们创建索引。我们唯一需要做的仅仅是选择一个索引名。这个名字必须是全部小写，不能以下划线开头，不能包含逗号。让我们使用`website`做为索引名。

`_type`

在应用中，我们使用对象表示一些“事物”，例如一个用户、一篇博客、一个评论，或者一封邮件。每个对象都属于一个类**(class)**，这个类定义了属性或与对象关联的数据。`user`类的对象可能包含姓名、性别、年龄和Email地址。

在关系型数据库中，我们经常将相同类的对象存储在一个表里，因为它们有着相同的结构。同理，在Elasticsearch中，我们使用相同类型**(type)**的文档表示相同的“事物”，因为他们的数据结构也是相同的。

每个类型**(type)**都有自己的映射**(mapping)**或者结构定义，就像传统数据库表中的列一样。所有类型下的文档被存储在同一个索引下，但是类型的映射**(mapping)**会告诉Elasticsearch不同的文档如何被索引。 我们将会在《映射》章节探讨如何定义和管理映射，但是现在我们将依赖ELasticsearch去自动处理数据结构。

`_type`的名字可以是大写或小写，不能包含下划线或逗号。我们将使用`blog`做为类型名。

`_id`

id仅仅是一个字符串，它与`_index`和`_type`组合时，就可以在Elasticsearch中唯一标识一个文档。当创建一个文档，你可以自定义`_id`，也可以让Elasticsearch帮你自动生成。

其它元数据

还有一些其它的元数据，我们将在《映射》章节探讨。使用上面提到的元素，我们已经可以在Elasticsearch中存储文档并通过ID检索——换言说，把Elasticsearch做为文档存储器使用了。

索引一个文档

文档通过 `index` API被索引——使数据可以被存储和搜索。但是首先我们需要决定文档所在。正如我们讨论的，文档通过其 `_index` 、 `_type` 、 `_id` 唯一确定。们可以自己提供一个 `_id` ，或者也使用 `index` API 为我们生成一个。

使用自己的ID

如果你的文档有自然的标识符（例如 `user_account` 字段或者其他值表示文档），你就可以提供自己的 `_id` ，使用这种形式的 `index` API：

```
PUT /{index}/{type}/{id}
{
  "field": "value",
  ...
}
```

例如我们的索引叫做 `“website”` ，类型叫做 `“blog”` ，我们选择的ID是 `“123”` ，那么这个索引请求就像这样：

```
PUT /website/blog/123
{
  "title": "My first blog entry",
  "text":  "Just trying this out...",
  "date":  "2014/01/01"
}
```

Elasticsearch的响应：

```
{
   "_index":    "website",
   "_type":     "blog",
   "_id":       "123",
   "_version":  1,
   "created":   true
}
```

响应指出请求的索引已经被成功创建，这个索引中包含 `_index` 、 `_type` 和 `_id` 元数据，以及一个新元素： `_version` 。

Elasticsearch中每个文档都有版本号，每当文档变化（包括删除）都会使 `_version` 增加。在《版本控制》章节中我们将探讨如何使用 `_version` 号确保你程序的一部分不会覆盖掉另一部分所做的更改。

自增ID

如果我们的数据没有自然ID，我们可以让Elasticsearch自动为我们生成。请求结构发生了变化： `PUT` 方法—— `“在这个URL中存储文档”` 变成了 `POST` 方法—— `"在这个文档下存储文档"` 。（译者注：原来是把文档存储到某个ID对应的空间，现在是把这个文档添加到某个 `_type` 下）。

URL现在只包含 `_index` 和 `_type` 两个字段：

```
POST /website/blog/
{
  "title": "My second blog entry",
  "text":  "Still trying this out...",
  "date":  "2014/01/01"
}
```

响应内容与刚才类似，只有 `_id` 字段变成了自动生成的值：

```
{
   "_index":    "website",
   "_type":     "blog",
   "_id":       "wM0OSFhDQXGZAWDf0-drSA",
   "_version":  1,
   "created":   true
}
```


自动生成的ID有22个字符长，URL-safe, Base64-encoded string universally unique identifiers, 或者叫 UUIDs。

检索文档

想要从Elasticsearch中获取文档，我们使用同样的 `_index` 、 `_type` 、 `_id` ，但是HTTP方法改为 `GET` ：

```
GET /website/blog/123?pretty
```

响应包含了现在熟悉的元数据节点，增加了 `_source` 字段，它包含了在创建索引时我们发送给Elasticsearch的原始文档。

```
{
  "_index" :   "website",
  "_type" :    "blog",
  "_id" :      "123",
  "_version" : 1,
  "found" :    true,
  "_source" :  {
      "title": "My first blog entry",
      "text":  "Just trying this out...",
      "date":  "2014/01/01"
  }
}
```

> **`pretty`**
>
> 在任意的查询字符串中增加 `pretty` 参数， 类似于上面的例子。会让Elasticsearch美化输出**(pretty-print)**JSON响应以便更加容易阅读。 `_source` 字段不会被美化，它的样子与我们输入的一致。

GET请求返回的响应内容包括 `{"found": true}` 。这意味着文档已经找到。如果我们请求一个不存在的文档，依旧会得到一个JSON，不过 `found` 值变成了 `false` 。

此外，HTTP响应状态码也会变成 `'404 Not Found'` 代替 `'200 OK'` 。我们可以在 `curl` 后加 `-i` 参数得到响应头：

```
curl -i -XGET http://localhost:9200/website/blog/124?pretty
```

现在响应类似于这样：

```
HTTP/1.1 404 Not Found
Content-Type: application/json; charset=UTF-8
Content-Length: 83

{
  "_index" : "website",
  "_type" :  "blog",
  "_id" :    "124",
  "found" :  false
}
```

检索文档的一部分

通常， `GET` 请求将返回文档的全部，存储在 `_source` 参数中。但是可能你感兴趣的字段只是 `title` 。请求个别字段可以使用 `_source` 参数。多个字段可以使用逗号分隔：

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

`_source` 字段现在只包含我们请求的字段，而且过滤了 `date` 字段：

```
{
  "_index" :   "website",
  "_type" :    "blog",
  "_id" :      "123",
  "_version" : 1,
  "exists" :   true,
  "_source" : {
      "title": "My first blog entry" ,
      "text":  "Just trying this out..."
  }
}
```

或者你只想得到 `_source` 字段而不要其他的元数据，你可以这样请求：

```
GET /website/blog/123/_source
```

它仅仅返回:

```
{
   "title": "My first blog entry",
   "text":  "Just trying this out...",
   "date":  "2014/01/01"
}
```

检查文档是否存在

如果你想做的只是检查文档是否存在——你对内容完全不感兴趣——使用 `HEAD` 方法来代替 `GET`。 `HEAD` 请求不会返回响应体，只有HTTP头：

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

Elasticsearch将会返回 `200 OK` 状态如果你的文档存在：

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=UTF-8
Content-Length: 0
```

如果不存在返回 `404 Not Found` ：

```
curl -i -XHEAD http://localhost:9200/website/blog/124
```

```
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=UTF-8
Content-Length: 0
```

当然，这只表示你在查询的那一刻文档不存在，但并不表示几毫秒后依旧不存在。另一个进程在这期间可能创建新文档。

更新整个文档

文档在Elasticsearch中是不可变的——我们不能修改他们。如果需要更新已存在的文档，我们可以使用《索引文档》章节提到的`index` API 重建索引*(reindex)* 或者替换掉它。

```
PUT /website/blog/123
{
  "title": "My first blog entry",
  "text":  "I am starting to get the hang of this...",
  "date":  "2014/01/02"
}
```

在响应中，我们可以看到Elasticsearch把`_version`增加了。

```
{
  "_index" :   "website",
  "_type" :    "blog",
  "_id" :      "123",
  "_version" : 2,
  "created":   false <1>
}
```

- <1> `created`标识为`false`因为同索引、同类型下已经存在同ID的文档。

在内部，Elasticsearch已经标记旧文档为删除并添加了一个完整的新文档。旧版本文档不会立即消失，但你也不能去访问它。Elasticsearch会在你继续索引更多数据时清理被删除的文档。

在本章的后面，我们将会在《局部更新》中探讨`update` API。这个API 似乎 允许你修改文档的局部，但事实上Elasticsearch遵循与之前所说完全相同的过程，这个过程如下：

1. 从旧文档中检索JSON
2. 修改它
3. 删除旧文档
4. 索引新文档

唯一的不同是`update` API完成这一过程只需要一个客户端请求既可，不再需要`get`和`index`请求了。

创建一个新文档

当索引一个文档，我们如何确定是完全创建了一个新的还是覆盖了一个已经存在的呢？

请记住 `_index` 、 `_type` 、 `_id` 三者唯一确定一个文档。所以要想保证文档是新加入的，最简单的方式是使用 `POST` 方法让Elasticsearch自动生成唯一 `_id` ：

```
POST /website/blog/
{ ... }
```

然而，如果想使用自定义的 `_id` ，我们必须告诉Elasticsearch应该在 `_index` 、 `_type` 、 `_id` 三者都不同时才接受请求。为了做到这点有两种方法，它们其实做的是同一件事情。你可以选择适合自己的方式：

第一种方法使用 `op_type` 查询参数：

```
PUT /website/blog/123?op_type=create
{ ... }
```

或者第二种方法是在URL后加 `/_create` 做为端点：

```
PUT /website/blog/123/_create
{ ... }
```

如果请求成功的创建了一个新文档，Elasticsearch将返回正常的元数据且响应状态码是 `201 Created` 。

另一方面，如果包含相同的 `_index` 、 `_type` 和 `_id` 的文档已经存在，Elasticsearch将返回 `409 Conflict` 响应状态码，错误信息类似如下：

```
{
  "error" : "DocumentAlreadyExistsException[[website][4] [blog][123]:
             document already exists]",
  "status" : 409
}
```

删除文档

删除文档的语法模式与之前基本一致，只不过要使用 `DELETE` 方法：

```
DELETE /website/blog/123
```

如果文档被找到，Elasticsearch将返回 `200 OK` 状态码和以下响应体。注意 `_version` 数字已经增加了。

```
{
  "found" :    true,
  "_index" :   "website",
  "_type" :    "blog",
  "_id" :      "123",
  "_version" : 3
}
```

如果文档未找到，我们将得到一个 `404 Not Found` 状态码，响应体是这样的：

```
{
  "found" :    false,
  "_index" :   "website",
  "_type" :    "blog",
  "_id" :      "123",
  "_version" : 4
}
```

尽管文档不存在—— `"found"` 的值是 `false` —— `_version` 依旧增加了。这是内部记录的一部分，它确保在多节点间不同操作可以有正确的顺序。

> 正如在《更新文档》一章中提到的，删除一个文档也不会立即从磁盘上移除，它只是被标记成已删除。Elasticsearch将会在你之后添加更多索引的时候才会在后台进行删除内容的清理。

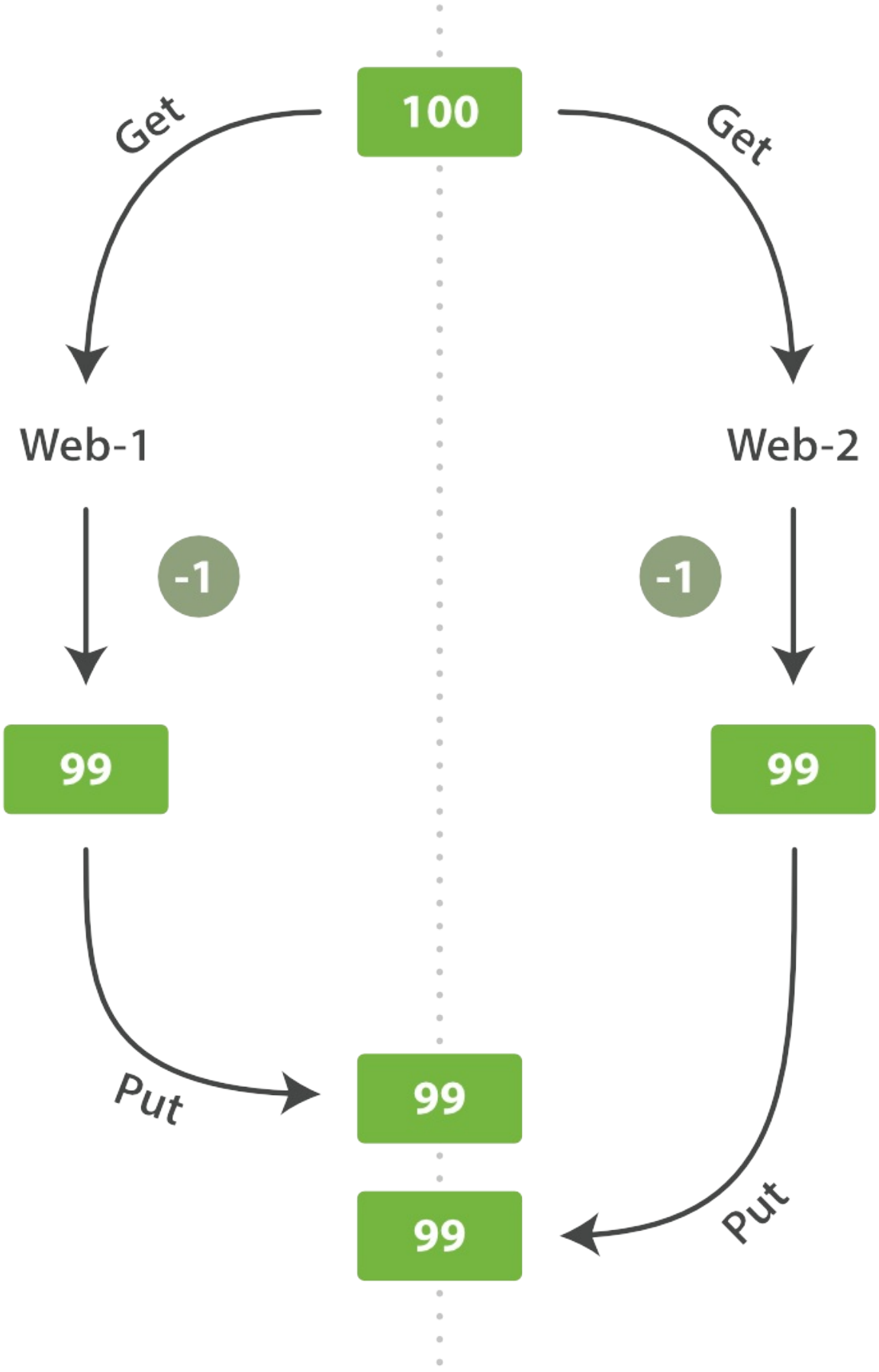
处理冲突

当使用 `index` API更新文档的时候，我们读取原始文档，做修改，然后将整个文档**(whole document)**一次性重新索引。最近的索引请求会生效——Elasticsearch中只存储最后被索引的任何文档。如果其他人同时也修改了这个文档，他们的修改将会丢失。

很多时候，这并不是一个问题。或许我们主要的数据存储在关系型数据库中，然后拷贝数据到Elasticsearch中只是为了可以用于搜索。或许两个人同时修改文档的机会很少。亦或者偶尔的修改丢失对于我们的工作来说并无大碍。

但有时丢失修改是一个很严重的问题。想象一下我们使用Elasticsearch存储大量在线商店的库存信息。每当销售一个商品，Elasticsearch中的库存就要减一。

一天，老板决定做一个促销。瞬间，我们每秒就销售了几个商品。想象两个同时运行的web进程，两者同时处理一件商品的订单：



`web_1` 让 `stock_count` 失效是因为 `web_2` 没有察觉到 `stock_count` 的拷贝已经过期（译者注： `web_1` 取数据，减一后更新

了 `stock_count`。可惜在 `web_1` 更新 `stock_count` 前它就拿到了数据，这个数据已经是过期的了，当 `web_2` 再回来更新 `stock_count` 时这个数字就是错的。这样就会造成看似卖了一件东西，其实是卖了两件，这个应该属于幻读。）。结果是我们认为自己确实还有更多的商品，最终顾客会因为销售给他们没有的东西而失望。

变化越是频繁，或读取和更新间的时间越长，越容易丢失我们的更改。

在数据库中，有两种通用的方法确保在并发更新时修改不丢失：

悲观并发控制（**Pessimistic concurrency control**）

这在关系型数据库中被广泛的使用，假设冲突的更改经常发生，为了解决冲突我们把访问区块化。典型的例子是在读一行数据前锁定这行，然后确保只有加锁的那个线程可以修改这行数据。

乐观并发控制（**Optimistic concurrency control**）：

被Elasticsearch使用，假设冲突不经常发生，也不区块化访问，然而，如果在读写过程中数据发生了变化，更新操作将失败。这时候由程序决定在失败后如何解决冲突。实际情况中，可以重新尝试更新，刷新数据（重新读取）或者直接反馈给用户。

乐观并发控制

Elasticsearch是分布式的。当文档被创建、更新或删除，文档的新版本会被复制到集群的其它节点。Elasticsearch即是同步的又是异步的，意思是这些复制请求都是平行发送的，并无序**(out of sequence)**的到达目的地。这就需要一种方法确保老版本的文档永远不会覆盖新的版本。

上文我们提到 `index`、 `get`、 `delete` 请求时，我们指出每个文档都有一个 `_version` 号码，这个号码在文档被改变时加一。Elasticsearch使用这个 `_version` 保证所有修改都被正确排序。当一个旧版本出现在新版本之后，它会被简单的忽略。

我们利用 `_version` 的这一优点确保数据不会因为修改冲突而丢失。我们可以指定文档的 `version` 来做想要的更改。如果那个版本号不是现在的，我们的请求就失败了。

Let's create a new blog post: 让我们创建一个新的博文：

```
PUT /website/blog/1/_create
{
  "title": "My first blog entry",
  "text":  "Just trying this out..."
}
```

响应体告诉我们这是一个新建的文档，它的 `_version` 是 `1`。现在假设我们要编辑这个文档：把数据加载到web表单中，修改，然后保存成新版本。

首先我们检索文档：

```
GET /website/blog/1
```

响应体包含相同的 `_version` 是 `1`

```
{
  "_index" :   "website",
  "_type" :    "blog",
  "_id" :      "1",
  "_version" : 1,
  "found" :    true,
  "_source" :  {
      "title": "My first blog entry",
      "text":  "Just trying this out..."
  }
```

```
}
```

现在，当我们通过重新索引文档保存修改时，我们这样指定了 `version` 参数：

```
PUT /website/blog/1?version=1 <1>
{
  "title": "My first blog entry",
  "text":  "Starting to get the hang of this..."
}
```

- <1> 我们只希望文档的 `_version` 是 `1` 时更新才生效。

This request succeeds, and the response body tells us that the `_version` has been incremented to `2`:

请求成功，响应体告诉我们 `_version` 已经增加到 `2` ：

```
{
  "_index":   "website",
  "_type":    "blog",
  "_id":      "1",
  "_version": 2
  "created":  false
}
```

然而，如果我们重新运行相同的索引请求，依旧指定 `version=1`，Elasticsearch将返回 `409 Conflict` 状态的HTTP响应。响应体类似这样：

```
{
  "error" : "VersionConflictEngineException[[website][2] [blog][1]:
            version conflict, current [2], provided [1]]",
  "status" : 409
}
```

这告诉我们当前 `_version` 是 `2`，但是我们指定想要更新的版本是 `1`。

我们需要做什么取决于程序的需求。我们可以告知用户其他人修改了文档，你应该在保存前再看一下。而对于上文提到的商品 `stock_count`，我们需要重新检索最新文档然后申请新的更改操作。

所有更新和删除文档的请求都接受 `version` 参数，它可以允许在你的代码中增加乐观锁控制。

使用外部版本控制系统

一种常见的结构是使用一些其他的数据库做为主数据库，然后使用Elasticsearch搜索数据，这意味着所有主数据库发生变化，就要将其拷贝到Elasticsearch中。如果有多个进程负责这些数据的同步，就会遇到上面提到的并发问题。

如果主数据库有版本字段——或一些类似于 `timestamp` 等可以用于版本控制的字段——是你就可以在Elasticsearch的查询字符串后面添加 `version_type=external` 来使用这些版本号。版本号必须是整数，大于零小于 `9.2e+18` ——Java中的正的 `long` 。

外部版本号与之前说的内部版本号在处理的时候有些不同。它不再检查 `_version` 是否与请求中指定的一致，而是检查是否小于指定的版本。如果请求成功，外部版本号就会被存储到 `_version` 中。

外部版本号不仅在索引和删除请求中指定，也可以在创建**(create)**新文档中指定。

例如，创建一个包含外部版本号 `5` 的新博客，我们可以这样做：

```
PUT /website/blog/2?version=5&version_type=external
```

```
{
  "title": "My first external blog entry",
  "text":  "Starting to get the hang of this..."
}
```

在响应中，我们能看到当前的 `_version` 号码是 `5` ：

```
{
  "_index":   "website",
  "_type":    "blog",
  "_id":      "2",
  "_version": 5,
  "created":  true
}
```

现在我们更新这个文档，指定一个新 `version` 号码为 `10` ：

```
PUT /website/blog/2?version=10&version_type=external
{
  "title": "My first external blog entry",
  "text":  "This is a piece of cake..."
}
```

请求成功的设置了当前 `_version` 为 `10` ：

```
{
  "_index":   "website",
  "_type":    "blog",
  "_id":      "2",
  "_version": 10,
  "created":  false
}
```

如果你重新运行这个请求，就会返回一个像之前一样的冲突错误，因为指定的外部版本号不大于当前在Elasticsearch中的版本。

文档局部更新

在《更新文档》一章，我们说了一种通过检索，修改，然后重建整文档的索引方法来更新文档。这是对的。然而，使用 `update` API，我们可以使用一个请求来实现局部更新，例如增加数量的操作。

我们也说过文档是不可变的——它们不能被更改，只能被替换。 `update` API必须遵循相同的规则。表面看来，我们似乎是局部更新了文档的位置，内部却是像我们之前说的一样简单的使用 `update` API处理相同的检索-修改-重建索引流程，我们也减少了其他进程可能导致冲突的修改。

最简单的 `update` 请求表单接受一个局部文档参数 `doc` ，它会合并到现有文档中——对象合并在一起，存在的标量字段被覆盖，新字段被添加。举个例子，我们可以使用以下请求为博客添加一个 `tags` 字段和一个 `views` 字段：

```
POST /website/blog/1/_update
{
   "doc" : {
      "tags" : [ "testing" ],
      "views": 0
   }
}
```

如果请求成功，我们将看到类似 `index` 请求的响应结果：

```
{
   "_index" :   "website",
   "_id" :      "1",
   "_type" :    "blog",
   "_version" : 3
}
```

检索文档文档显示被更新的 `_source` 字段：

```
{
   "_index":    "website",
   "_type":     "blog",
   "_id":       "1",
   "_version":  3,
   "found":     true,
   "_source": {
      "title":  "My first blog entry",
      "text":   "Starting to get the hang of this...",
      "tags": [ "testing" ], <1>
      "views":  0 <1>
   }
}
```

- <1> 我们新添加的字段已经被添加到 `_source` 字段中。

使用脚本局部更新

> ### 使用Groovy脚本
>
> 这时候当API不能满足要求时，Elasticsearch允许你使用脚本实现自己的逻辑。脚本支持非常多的API，例如搜索、排序、聚合和文档更新。脚本可以通过请求的一部分、检索特殊的 `.scripts` 索引或者从磁盘加载方式执行。
>
> 默认的脚本语言是Groovy，一个快速且功能丰富的脚本语言，语法类似于Javascript。它在一个沙盒**(sandbox)**中运行，以防止恶意用户毁坏Elasticsearch或攻击服务器。
>
> 你可以在《脚本参考文档》中获得更多信息。

脚本能够使用 `update` API改变 `_source` 字段的内容，它在脚本内部以 `ctx._source` 表示。例如，我们可以使用脚本增加博客的 `views` 数量：

```
POST /website/blog/1/_update
{
   "script" : "ctx._source.views+=1"
}
```

我们还可以使用脚本增加一个新标签到 `tags` 数组中。在这个例子中，我们定义了一个新标签做为参数而不是硬编码在脚本里。这允许Elasticsearch未来可以重复利用脚本，而不是在想要增加新标签时必须每次编译新脚本：

```
POST /website/blog/1/_update
{
   "script" : "ctx._source.tags+=new_tag",
   "params" : {
      "new_tag" : "search"
   }
}
```

获取最后两个有效请求的文档：

```
{
   "_index":    "website",
   "_type":     "blog",
   "_id":       "1",
   "_version":  5,
   "found":     true,
   "_source": {
      "title":  "My first blog entry",
      "text":   "Starting to get the hang of this...",
      "tags":  ["testing", "search"], <1>
      "views":  1 <2>
   }
}
```

- <1> `search` 标签已经被添加到 `tags` 数组。
- <2> `views` 字段已经被增加。

通过设置 `ctx.op` 为 `delete` 我们可以根据内容删除文档：

```
POST /website/blog/1/_update
{
   "script" : "ctx.op = ctx._source.views == count ? 'delete' : 'none'",
    "params" : {
        "count": 1
    }
}
```

更新可能不存在的文档

想象我们要在Elasticsearch中存储浏览量计数器。每当有用户访问页面，我们增加这个页面的浏览量。但如果这是个新页面，我们并不确定这个计数器存在与否。当我们试图更新一个不存在的文档，更新将失败。

在这种情况下，我们可以使用 `upsert` 参数定义文档来使其不存在时被创建。

```
POST /website/pageviews/1/_update
{
   "script" : "ctx._source.views+=1",
   "upsert": {
       "views": 1
   }
}
```

第一次执行这个请求，`upsert`值被索引为一个新文档，初始化`views`字段为`1`.接下来文档已经存在，所以`script`被更新代替，增加`views`数量。

更新和冲突

这这一节的介绍中，我们介绍了如何在检索**(retrieve)**和重建索引**(reindex)**中保持更小的窗口，如何减少冲突性变更发生的概率，不过这些无法被完全避免，像一个其他进程在`update`进行重建索引时修改了文档这种情况依旧可能发生。

为了避免丢失数据，`update` API在检索**(retrieve)**阶段检索文档的当前`_version`，然后在重建索引**(reindex)**阶段通过`index`请求提交。如果其他进程在检索**(retrieve)**和重加索引**(reindex)**阶段修改了文档，`_version`将不能被匹配，然后更新失败。

对于多用户的局部更新，文档被修改了并不要紧。例如，两个进程都要增加页面浏览量，增加的顺序我们并不关心——如果冲突发生，我们唯一要做的仅仅是重新尝试更新既可。

这些可以通过`retry_on_conflict`参数设置重试次数来自动完成，这样`update`操作将会在发生错误前重试——这个值默认为0。

```
POST /website/pageviews/1/_update?retry_on_conflict=5 <1>
{
   "script" : "ctx._source.views+=1",
   "upsert": {
       "views": 0
   }
}
```

- <1> 在错误发生前重试更新5次

这适用于像增加计数这种顺序无关的操作，但是还有一种顺序非常重要的情况。例如`index` API，使用“保留最后更新**(last-write-wins)**”的`update` API，但它依旧接受一个`version`参数以允许你使用乐观并发控制**(optimistic concurrency control)**来指定你要更细文档的版本。

检索多个文档

像Elasticsearch一样，检索多个文档依旧非常快。合并多个请求可以避免每个请求单独的网络开销。如果你需要从Elasticsearch中检索多个文档，相对于一个一个的检索，更快的方式是在一个请求中使用**multi-get**或者 `mget` API。

`mget` API参数是一个 `docs` 数组，数组的每个节点定义一个文档的 `_index` 、 `_type` 、 `_id` 元数据。如果你只想检索一个或几个确定的字段，也可以定义一个 `_source` 参数：

```
GET /_mget
{
   "docs" : [
      {
         "_index" : "website",
         "_type" :  "blog",
         "_id" :    2
      },
      {
         "_index" : "website",
         "_type" :  "pageviews",
         "_id" :    1,
         "_source": "views"
      }
   ]
}
```

响应体也包含一个 `docs` 数组，每个文档还包含一个响应，它们按照请求定义的顺序排列。每个这样的响应与单独使用 **`get` request**响应体相同：

```
{
   "docs" : [
      {
         "_index" :   "website",
         "_id" :      "2",
         "_type" :    "blog",
         "found" :    true,
         "_source" : {
            "text" :  "This is a piece of cake...",
            "title" : "My first external blog entry"
         },
         "_version" : 10
      },
      {
         "_index" :   "website",
         "_id" :      "1",
         "_type" :    "pageviews",
         "found" :    true,
         "_version" : 2,
         "_source" : {
            "views" : 2
         }
      }
   ]
}
```

如果你想检索的文档在同一个 `_index` 中（甚至在同一个 `_type` 中），你就可以在URL中定义一个默认的 `/_index` 或者 `/_index/_type` 。

你依旧可以在单独的请求中使用这些值：

```
GET /website/blog/_mget
{
   "docs" : [
      { "_id" : 2 },
      { "_type" : "pageviews", "_id" :   1 }
   ]
}
```

事实上，如果所有文档具有相同 `_index` 和 `_type`，你可以通过简单的 `ids` 数组来代替完整的 `docs` 数组：

```
GET /website/blog/_mget
{
   "ids" : [ "2", "1" ]
}
```

注意到我们请求的第二个文档并不存在。我们定义了类型为 `blog`，但是ID为 `1` 的文档类型为 `pageviews`。这个不存在的文档会在响应体中被告知。

```
{
  "docs" : [
    {
      "_index" :   "website",
      "_type" :    "blog",
      "_id" :      "2",
      "_version" : 10,
      "found" :    true,
      "_source" : {
        "title":   "My first external blog entry",
        "text":    "This is a piece of cake..."
      }
    },
    {
      "_index" :   "website",
      "_type" :    "blog",
      "_id" :      "1",
      "found" :    false  <1>
    }
  ]
}
```

- <1> 这个文档不存在

事实上第二个文档不存在并不影响第一个文档的检索。每个文档的检索和报告都是独立的。

> 注意：
>
> 尽管前面提到有一个文档没有被找到，但HTTP请求状态码还是 `200`。事实上，就算所有文档都找不到，请求也还是返回 `200`，原因是 `mget` 请求本身成功了。如果想知道每个文档是否都成功了，你需要检查 `found` 标志。

更省时的批量操作

就像 `mget` 允许我们一次性检索多个文档一样，`bulk` API允许我们使用单一请求来实现多个文档的 `create`、`index`、`update` 或 `delete`。这对索引类似于日志活动这样的数据流非常有用，它们可以以成百上千的数据为一个批次按序进行索引。

`bulk` 请求体如下，它有一点不同寻常：

```
{ action: { metadata }}\n
{ request body        }\n
{ action: { metadata }}\n
{ request body        }\n
...
```

这种格式类似于用 `"\n"` 符号连接起来的一行一行的JSON文档流**(stream)**。两个重要的点需要注意：

- 每行必须以 `"\n"` 符号结尾，包括最后一行。这些都是作为每行有效的分离而做的标记。
- 每一行的数据不能包含未被转义的换行符，它们会干扰分析——这意味着JSON不能被美化打印。

> 提示:
>
> 在《批量格式》一章我们介绍了为什么 `bulk` API使用这种格式。

action/metadata这一行定义了文档行为**(what action)**发生在哪个文档**(which document)**之上。

行为**(action)**必须是以下几种：

行为	解释
`create`	当文档不存在时创建之。详见《创建文档》
`index`	创建新文档或替换已有文档。见《索引文档》和《更新文档》
`update`	局部更新文档。见《局部更新》
`delete`	删除一个文档。见《删除文档》

在索引、创建、更新或删除时必须指定文档的 `_index`、`_type`、`_id` 这些元数据**(metadata)**。

例如删除请求看起来像这样：

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

请求体**(request body)**由文档的 `_source` 组成——文档所包含的一些字段以及其值。它被 `index` 和 `create` 操作所必须，这是有道理的：你必须提供文档用来索引。

这些还被 `update` 操作所必需，而且请求体的组成应该与 `update` API（`doc`，`upsert`，`script` 等等）一致。删除操作不需要请求体**(request body)**。

```
{ "create":  { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "My first blog post" }
```

如果定义 `_id`，ID将会被自动创建：

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

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

为了将这些放在一起，`bulk` 请求表单是这样的：

```
POST /_bulk
{ "delete": { "_index": "website", "_type": "blog", "_id": "123" }} <1>
{ "create": { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "My first blog post" }
{ "index":  { "_index": "website", "_type": "blog" }}
{ "title":    "My second blog post" }
{ "update": { "_index": "website", "_type": "blog", "_id": "123", "_retry_on_conflict" : 3} }
{ "doc" : {"title" : "My updated blog post"} } <2>
```

- <1> 注意 `delete` 行为**(action)**没有请求体，它紧接着另一个行为**(action)**
- <2> 记得最后一个换行符

Elasticsearch响应包含一个 `items` 数组，它罗列了每一个请求的结果，结果的顺序与我们请求的顺序相同：

```
{
   "took": 4,
   "errors": false, <1>
   "items": [
      {  "delete": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "_version": 2,
            "status":   200,
            "found":    true
      }},
      {  "create": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "_version": 3,
            "status":   201
      }},
      {  "create": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "EiwfApScQiiy7TIKFxRCTw",
            "_version": 1,
            "status":   201
      }},
      {  "update": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "_version": 4,
            "status":   200
      }}
   ]
}}
```

- <1> 所有子请求都成功完成。

每个子请求都被独立的执行，所以一个子请求的错误并不影响其它请求。如果任何一个请求失败，顶层的 `error` 标记将被设置为 `true`，然后错误的细节将在相应的请求中被报告：

```
POST /_bulk
{ "create": { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "Cannot create - it already exists" }
{ "index":  { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "But we can update it" }
```

响应中我们将看到 `create` 文档 `123` 失败了，因为文档已经存在，但是后来的在 `123` 上执行的 `index` 请求成功了：

```
{
   "took": 3,
   "errors": true, <1>
   "items": [
      {  "create": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "status":   409, <2>
            "error":    "DocumentAlreadyExistsException <3>
                        [[website][4] [blog][123]:
                        document already exists]"
      }},
      {  "index": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "_version": 5,
            "status":   200 <4>
      }}
   ]
}
```

- <1> 一个或多个请求失败。
- <2> 这个请求的HTTP状态码被报告为 `409 CONFLICT` 。
- <3> 错误消息说明了什么请求错误。
- <4> 第二个请求成功了，状态码是 `200 OK` 。

这些说明 `bulk` 请求不是原子操作——它们不能实现事务。每个请求操作时分开的，所以每个请求的成功与否不干扰其它操作。

不要重复

你可能在同一个 `index` 下的同一个 `type` 里批量索引日志数据。为每个文档指定相同的元数据是多余的。就像 `mget` API， `bulk` 请求也可以在URL中使用 `/_index` 或 `/_index/_type` ：

```
POST /website/_bulk
{ "index": { "_type": "log" }}
{ "event": "User logged in" }
```

你依旧可以覆盖元数据行的 `_index` 和 `_type` ，在没有覆盖时它会使用URL中的值作为默认值：

```
POST /website/log/_bulk
{ "index": {}}
{ "event": "User logged in" }
{ "index": { "_type": "blog" }}
{ "title": "Overriding the default type" }
```

多大才算太大？

整个批量请求需要被加载到接受我们请求节点的内存里，所以请求越大，给其它请求可用的内存就越小。有一个最佳的 `bulk` 请求大小。超过这个大小，性能不再提升而且可能降低。

最佳大小，当然并不是一个固定的数字。它完全取决于你的硬件、你文档的大小和复杂度以及索引和搜索的负载。幸运的是，这个最佳点**(sweetspot)**还是容易找到的：

试着批量索引标准的文档，随着大小的增长，当性能开始降低，说明你每个批次的大小太大了。开始的数量可以在1000~5000个文档之间，如果你的文档非常大，可以使用较小的批次。

通常着眼于你请求批次的物理大小是非常有用的。一千个1kB的文档和一千个1MB的文档大不相同。一个好的批次最好保持在5-15MB大小间。

结语

现在你知道如何把Elasticsearch当作一个分布式的文件存储了。你可以存储、更新、检索和删除它们，而且你知道如何安全的进行这一切。这确实非常非常有用，尽管我们还没有看到更多令人激动的特性，例如如何在文档内搜索。但让我们首先讨论下如何在分布式环境中安全的管理你的文档相关的内部流程。

分布式文档存储

在上一章，我们看到了将数据放入索引然后检索它们的所有方法。不过我们有意略过了许多关于数据是如何在集群中分布和获取的相关技术细节。这种使用和细节分离是刻意为之的——你不需要知道数据在Elasticsearch如何分布它就会很好的工作。

这一章我们深入这些内部细节来帮助你更好的理解数据是如何在分布式系统中存储的。

> 注意：
>
> 下面的信息只是出于兴趣阅读，你不必为了使用Elasticsearch而弄懂和记住所有的细节。讨论的这些选项只提供给高级用户。
>
> 阅读这一部分只是让你了解下系统如何工作，并让你知道这些信息以备以后参考，所以不要被细节吓到。

路由文档到分片

当你索引一个文档，它被存储在单独一个主分片上。Elasticsearch是如何知道文档属于哪个分片的呢？当你创建一个新文档，它是如何知道是应该存储在分片1还是分片2上的呢？

进程不能是随机的，因为我们将来要检索文档。事实上，它根据一个简单的算法决定：

```
shard = hash(routing) % number_of_primary_shards
```

`routing` 值是一个任意字符串，它默认是 `_id` 但也可以自定义。这个 `routing` 字符串通过哈希函数生成一个数字，然后除以主切片的数量得到一个余数**(remainder)**，余数的范围永远是 `0` 到 `number_of_primary_shards - 1`，这个数字就是特定文档所在的分片。

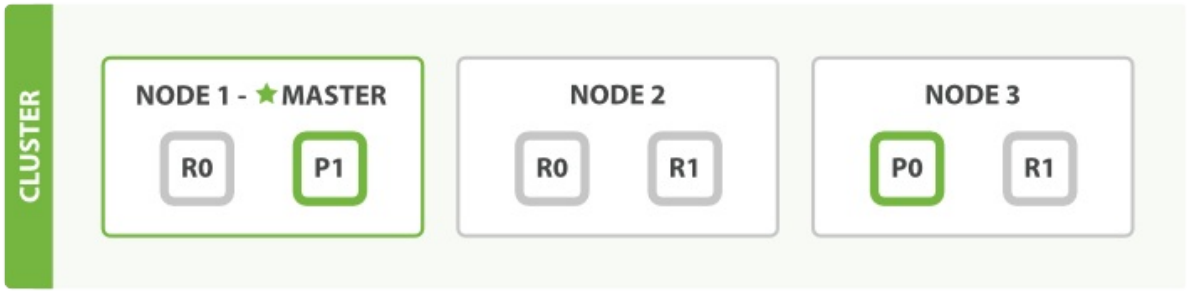
这也解释了为什么主分片的数量只能在创建索引时定义且不能修改：如果主分片的数量在未来改变了，所有先前的路由值就失效了，文档也就永远找不到了。

> 有时用户认为固定数量的主分片会让之后的扩展变得很困难。现实中，有些技术会在你需要的时候让扩展变得容易。我们将在《扩展》章节讨论。

所有的文档API（ `get` 、 `index` 、 `delete` 、 `bulk` 、 `update` 、 `mget` ）都接收一个 `routing` 参数，它用来自定义文档到分片的映射。自定义路由值可以确保所有相关文档——例如属于同一个人的文档——被保存在同一分片上。我们将在《扩展》章节说明你为什么需要这么做。

主分片和复制分片如何交互

为了阐述意图，我们假设有三个节点的集群。它包含一个叫做 `bblogs` 的索引并拥有两个主分片。每个主分片有两个复制分片。相同的分片不会放在同一个节点上，所以我们的集群是这样的：

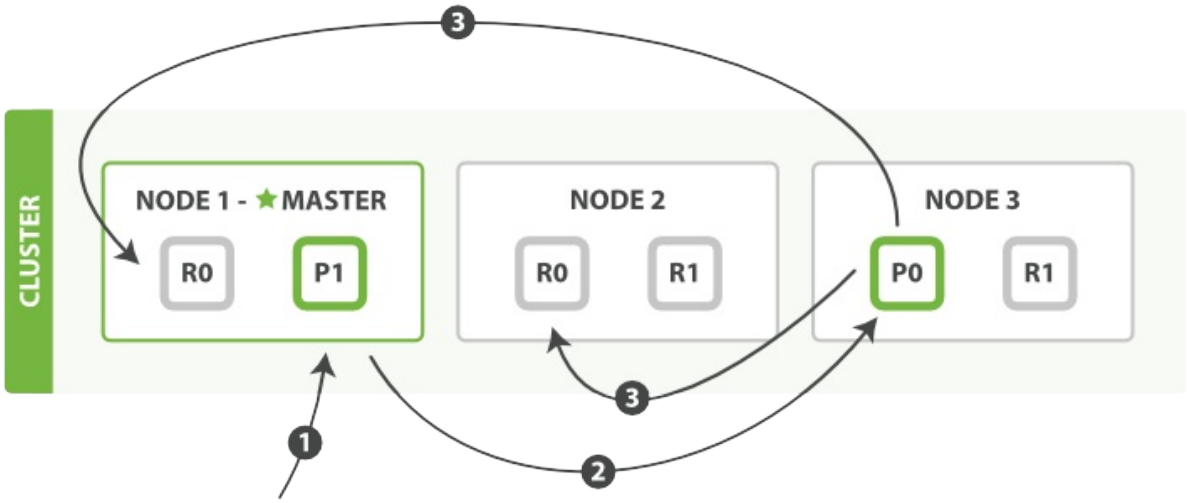


我们能够发送请求给集群中任意一个节点。每个节点都有能力处理任意请求。每个节点都知道任意文档所在的节点，所以也可以将请求转发到需要的节点。下面的例子中，我们将发送所有请求给 `Node 1`，这个节点我们将会称之为请求节点 **(requesting node)**

> 提示：
>
> 当我们发送请求，最好的做法是循环通过所有节点请求，这样可以平衡负载。

新建、索引和删除文档

新建、索引和删除请求都是写**(write)**操作，它们必须在主分片上成功完成才能复制到相关的复制分片上。



下面我们罗列在主分片和复制分片上成功新建、索引或删除一个文档必要的顺序步骤：

1. 客户端给 `Node 1` 发送新建、索引或删除请求。
2. 节点使用文档的 `_id` 确定文档属于分片 `0` 。它转发请求到 `Node 3` ，分片 `0` 位于这个节点上。
3. `Node 3` 在主分片上执行请求，如果成功，它转发请求到相应的位于 `Node 1` 和 `Node 2` 的复制节点上。当所有的复制节点报告成功， `Node 3` 报告成功到请求的节点，请求的节点再报告给客户端。

客户端接收到成功响应的时候，文档的修改已经被应用于主分片和所有的复制分片。你的修改生效了。

有很多可选的请求参数允许你更改这一过程。你可能想牺牲一些安全来提高性能。这一选项很少使用因为Elasticsearch已经足够快，不过为了内容的完整我们将做一些阐述。

`replication`

复制默认的值是 `sync` 。这将导致主分片得到复制分片的成功响应后才返回。

如果你设置 `replication` 为 `async` ，请求在主分片上被执行后就会返回给客户端。它依旧会转发请求给复制节点，但你将不知道复制节点成功与否。

上面的这个选项不建议使用。默认的 `sync` 复制允许Elasticsearch强制反馈传输。 `async` 复制可能会因为在不等待其它分片就绪的情况下发送过多的请求而使Elasticsearch过载。

`consistency`

默认主分片在尝试写入时需要规定数量**(quorum)**或过半的分片（可以是主节点或复制节点）可用。这是防止数据被写入到错的网络分区。规定的数量计算公式如下：

```
int( (primary + number_of_replicas) / 2 ) + 1
```

`consistency` 允许的值为 `one` （只有一个主分片）， `all` （所有主分片和复制分片）或者默认的 `quorum` 或过半分片。

注意 `number_of_replicas` 是在索引中的的设置，用来定义复制分片的数量，而不是现在活动的复制节点的数量。如果你定义了索引有3个复制节点，那规定数量是：

```
int( (primary + 3 replicas) / 2 ) + 1 = 3
```

但如果你只有2个节点，那你的活动分片不够规定数量，也就不能索引或删除任何文档。

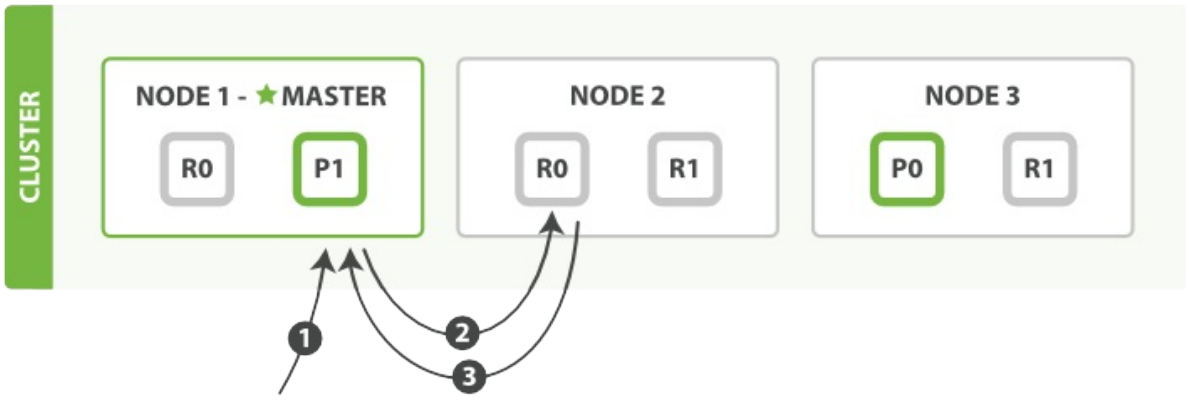
`timeout`

当分片副本不足时会怎样？Elasticsearch会等待更多的分片出现。默认等待一分钟。如果需要，你可以设置 `timeout` 参数让它终止的更早： `100` 表示100毫秒， `30s` 表示30秒。

> 注意：
>
> 新索引默认有 `1` 个复制分片，这意味着为了满足 `quorum` 的要求需要两个活动的分片。当然，这个默认设置将阻止我们在单一节点集群中进行操作。为了避开这个问题，规定数量只有在 `number_of_replicas` 大于一时才生效。

检索文档

文档能够从主分片或任意一个复制分片被检索。



下面我们罗列在主分片或复制分片上检索一个文档必要的顺序步骤：

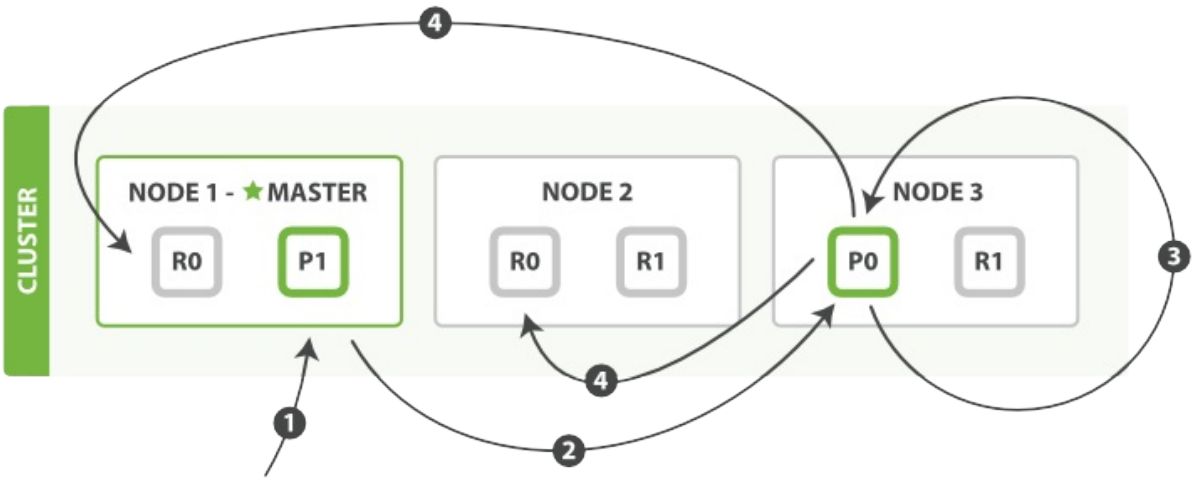
1. 客户端给 `Node 1` 发送get请求。
2. 节点使用文档的 `_id` 确定文档属于分片 `0` 。分片 `0` 对应的复制分片在三个节点上都有。此时，它转发请求到 `Node 2` 。
3. `Node 2` 返回endangered给 `Node 1` 然后返回给客户端。

对于读请求，为了平衡负载，请求节点会为每个请求选择不同的分片——它会循环所有分片副本。

可能的情况是，一个被索引的文档已经存在于主分片上却还没来得及同步到复制分片上。这时复制分片会报告文档未找到，主分片会成功返回文档。一旦索引请求成功返回给用户，文档则在主分片和复制分片都是可用的。

局部更新文档

`update` API 结合了之前提到的读和写的模式。



下面我们罗列执行局部更新必要的顺序步骤：

1. 客户端给 `Node 1` 发送更新请求。
2. 它转发请求到主分片所在节点 `Node 3` 。
3. `Node 3` 从主分片检索出文档，修改 `_source` 字段的JSON，然后在主分片上重建索引。如果有其他进程修改了文档，它以 `retry_on_conflict` 设置的次数重复步骤3，都未成功则放弃。
4. 如果 `Node 3` 成功更新文档，它同时转发文档的新版本到 `Node 1` 和 `Node 2` 上的复制节点以重建索引。当所有复制节点报告成功， `Node 3` 返回成功给请求节点，然后返回给客户端。

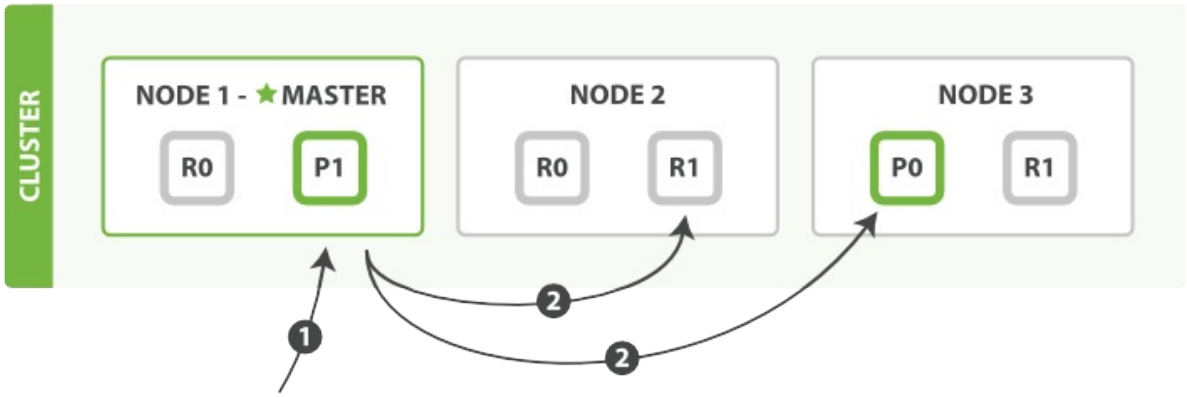
`update` API还接受《新建、索引和删除》章节提到的 `routing` 、 `replication` 、 `consistency` 和 `timout` 参数。

> ## 基于文档的复制
>
> 当主分片转发更改给复制分片时，并不是转发更新请求，而是转发整个文档的新版本。记住这些修改转发到复制节点是异步的，它们并不能保证到达的顺序与发送相同。如果Elasticsearch转发的仅仅是修改请求，修改的顺序可能是错误的，那得到的就是个损坏的文档。

多文档模式

`mget` 和 `bulk` API与单独的文档类似。差别是请求节点知道每个文档所在的分片。它把多文档请求拆成每个分片的对文档请求，然后转发每个参与的节点。

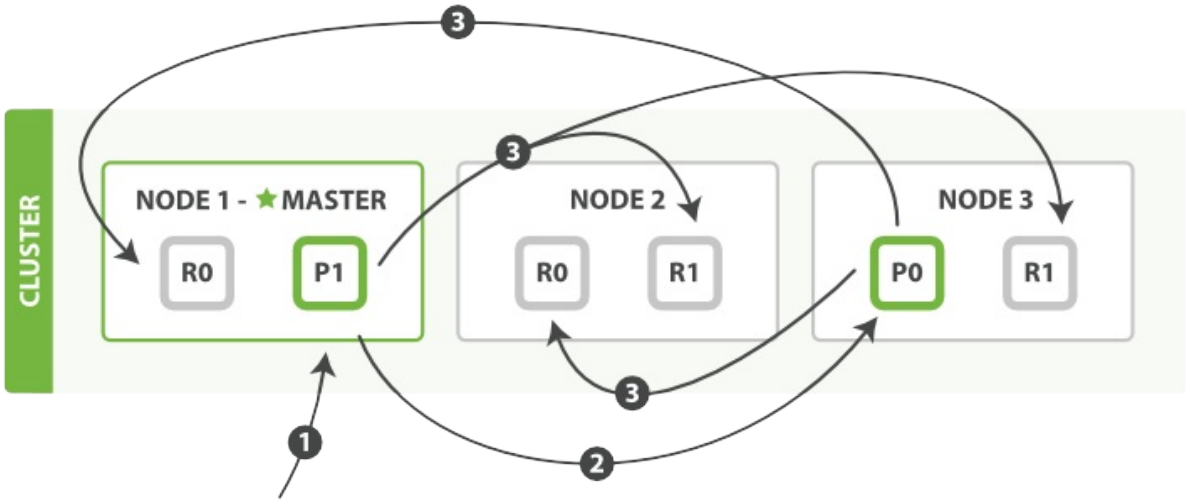
一旦接收到每个节点的应答，然后整理这些响应组合为一个单独的响应，最后返回给客户端。



下面我们将罗列通过一个 `mget` 请求检索多个文档的顺序步骤：

1. 客户端向 `Node 1` 发送 `mget` 请求。
2. `Node 1` 为每个分片构建一个多条数据检索请求，然后转发到这些请求所需的主分片或复制分片上。当所有回复被接收， `Node 1` 构建响应并返回给客户端。

`routing` 参数可以被 `docs` 中的每个文档设置。



下面我们将罗列使用一个 `bulk` 执行多个 `create` 、 `index` 、 `delete` 和 `update` 请求的顺序步骤：

1. 客户端向 `Node 1` 发送 `bulk` 请求。
2. `Node 1` 为每个分片构建批量请求，然后转发到这些请求所需的主分片上。
3. 主分片一个接一个的按序执行操作。当一个操作执行完，主分片转发新文档（或者删除部分）给对应的复制节点，然后执行下一个操作。复制节点为报告所有操作完成， 节点报告给请求节点，请求节点整理响应并返回给客户端。

`bulk` API还可以在最上层使用 `replication` 和 `consistency` 参数， `routing` 参数则在每个请求的元数据中使用。

为什么是奇怪的格式？

当我们在《批量》一章中学习了批量请求后，你可能会问："为什么 `bulk` API需要带换行符的奇怪格式，而不是像 `mget` API一样使用JSON数组？"

为了回答这个问题，我们需要简单的介绍一下背景：

批量中每个引用的文档属于不同的主分片，每个分片可能被分布于集群中的某个节点上。这意味着批量中的每个操作**(action)**需要被转发到对应的分片和节点上。

如果每个单独的请求被包装到JSON数组中，那意味着我们需要：

- 解析JSON为数组（包括文档数据，可能非常大）
- 检查每个请求决定应该到哪个分片上
- 为每个分片创建一个请求的数组
- 序列化这些数组为内部传输格式
- 发送请求到每个分片

这可行，但需要大量的RAM来承载本质上相同的数据，还要创建更多的数据结构使得JVM花更多的时间执行垃圾回收。

取而代之的，Elasticsearch则是从网络缓冲区中一行一行的直接读取数据。它使用换行符识别和解析**action/metadata**行，以决定哪些分片来处理这个请求。

这些行请求直接转发到对应的分片上。这些没有冗余复制，没有多余的数据结构。整个请求过程使用最小的内存在进行。

搜索——基本的工具

到目前为止，我们已经学会了如何使用elasticsearch作为一个简单的NoSQL风格的分布式文件存储器——我们可以将一个JSON文档扔给Elasticsearch，也可以根据ID检索它们。但Elasticsearch真正强大之处在于可以从混乱的数据中找出有意义的信息——从大数据到全面的信息。

这也是为什么我们使用结构化的JSON文档，而不是无结构的二进制数据。Elasticsearch不只会存储**(store)**文档，也会索引**(indexes)**文档内容来使之可以被搜索。

每个文档里的字段都会被索引并被查询。而且不仅如此。在简单查询时，Elasticsearch可以使用所有的索引，以非常快的速度返回结果。这让你永远不必考虑传统数据库的一些东西。

A *search* can be: 搜索**(search)**可以：

- 在类似于 `gender` 或者 `age` 这样的字段上使用结构化查询， `join_date` 这样的字段上使用排序，就像SQL的结构化查询一样。
- 全文检索，可以使用所有字段来匹配关键字，然后按照关联性**(relevance)**排序返回结果。
- 或者结合以上两条。

很多搜索都是开箱即用的，为了充分挖掘Elasticsearch的潜力，你需要理解以下三个概念：

概念	解释
映射**(Mapping)**	数据在每个字段中的解释说明
分析**(Analysis)**	全文是如何处理的可以被搜索的
领域特定语言查询**(Query DSL)**	Elasticsearch使用的灵活的、强大的查询语言

以上提到的每个点都是一个巨大的话题，我们将在《深入搜索》一章阐述它们。本章节我们将介绍这三点的一些基本概念——仅仅帮助你大致了解搜索是如何工作的。

我们将使用最简单的形式开始介绍 `search` API.

> ### 测试数据
>
> 本章节测试用的数据可以在这里被找到https://gist.github.com/clintongormley/8579281
>
> 你可以把这些命令复制到终端中执行以便可以实践本章的例子。

空搜索

最基本的搜索API表单是空搜索**(empty search)**，它没有指定任何的查询条件，只返回集群索引中的所有文档：

```
GET /_search
```

响应内容（为了编辑简洁）类似于这样：

```
{
   "hits" : {
      "total" :       14,
      "hits" : [
        {
          "_index":   "us",
          "_type":    "tweet",
          "_id":      "7",
          "_score":   1,
          "_source": {
             "date":    "2014-09-17",
             "name":    "John Smith",
             "tweet":   "The Query DSL is really powerful and flexible",
             "user_id": 2
          }
       },
        ... 9 RESULTS REMOVED ...
      ],
      "max_score" :   1
   },
   "took" :           4,
   "_shards" : {
      "failed" :      0,
      "successful" :  10,
      "total" :       10
   },
   "timed_out" :      false
}
```

hits

响应中最重要的部分是 `hits`，它包含了 `total` 字段来表示匹配到的文档总数，`hits` 数组还包含了匹配到的前10条数据。

`hits` 数组中的每个结果都包含 `_index`、`_type` 和文档的 `_id` 字段，被加入到 `_source` 字段中这意味着在搜索结果中我们将可以直接使用全部文档。这不像其他搜索引擎只返回文档ID，需要你单独去获取文档。

每个节点都有一个 `_score` 字段，这是相关性得分**(relevance score)**，它衡量了文档与查询的匹配程度。默认的，返回的结果中关联性最大的文档排在首位；这意味着，它是按照 `_score` 降序排列的。这种情况下，我们没有指定任何查询，所以所有文档的相关性是一样的，因此所有结果的 `_score` 都是取得一个中间值 `1`

`max_score` 指的是所有文档匹配查询中 `_score` 的最大值。

took

`took` 告诉我们整个搜索请求花费的毫秒数。

shards

`_shards` 节点告诉我们参与查询的分片数（`total` 字段），有多少是成功的（`successful` 字段），有多少的是失败的（`failed` 字段）。通常我们不希望分片失败，不过这个有可能发生。如果我们遭受一些重大的故障导致主分片和复制分片都故障，那这个分片的数据将无法响应给搜索请求。这种情况下，Elasticsearch将报告分片 `failed`，但仍将继续返回剩余分片

上的结果。

timeout

time_out 值告诉我们查询超时与否。一般的，搜索请求不会超时。如果响应速度比完整的结果更重要，你可以定义 timeout 参数为 10 或者 10ms （10毫秒），或者 1s （1秒）

```
GET /_search?timeout=10ms
```

Elasticsearch将返回在请求超时前收集到的结果。

超时不是一个断路器（circuit breaker）（译者注：关于断路器的理解请看警告）。

> ### 警告
>
> 需要注意的是 timeout 不会停止执行查询，它仅仅告诉你目前顺利返回结果的节点然后关闭连接。在后台，其他分片可能依旧执行查询，尽管结果已经被发送。
>
> 使用超时是因为对于你的业务需求（译者注：SLA，Service-Level Agreement服务等级协议，在此我翻译为业务需求）来说非常重要，而不是因为你想中断执行长时间运行的查询。

多索引和多类别

你注意到空搜索的结果中不同类型的文档—— `user` 和 `tweet` ——来自于不同的索引—— `us` 和 `gb` 。

通过限制搜索的不同索引或类型，我们可以在集群中跨所有文档搜索。Elasticsearch转发搜索请求到集群中平行的主分片或每个分片的复制分片上，收集结果后选择顶部十个返回给我们。

通常，当然，你可能想搜索一个或几个自定的索引或类型，我们能通过定义URL中的索引或类型达到这个目的，像这样：

`/_search`

在所有索引的所有类型中搜索

`/gb/_search`

在索引 `gb` 的所有类型中搜索

`/gb,us/_search`

在索引 `gb` 和 `us` 的所有类型中搜索

`/g*,u*/_search`

在以 `g` 或 `u` 开头的索引的所有类型中搜索

`/gb/user/_search`

在索引 `gb` 的类型 `user` 中搜索

`/gb,us/user,tweet/_search`

在索引 `gb` 和 `us` 的类型为 `user` 和 `tweet` 中搜索

`/_all/user,tweet/_search`

在所有索引的 `user` 和 `tweet` 中搜索 search types `user` and `tweet` in all indices

当你搜索包含单一索引时，Elasticsearch转发搜索请求到这个索引的主分片或每个分片的复制分片上，然后聚集每个分片的结果。搜索包含多个索引也是同样的方式——只不过或有更多的分片被关联。

> ## 重要
>
> 搜索一个索引有5个主分片和5个索引各有一个分片事实上是一样的。

接下来，你将看到这些简单的情况如何灵活的扩展以适应你需求的变更。

分页

《空搜索》一节告诉我们在集群中有14个文档匹配我们的（空）搜索语句。单数只有10个文档在 `hits` 数组中。我们如何看到其他文档？

和SQL使用 `LIMIT` 关键字返回只有一页的结果一样，Elasticsearch接受 `from` 和 `size` 参数：

`size`：果数，默认 `10`

`from`：跳过开始的结果数，默认 `0`

如果你想每页显示5个结果，页码从1到3，那请求如下：

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

应该当心分页太深或者一次请求太多的结果。结果在返回前会被排序。但是记住一个搜索请求常常涉及多个分片。每个分片生成自己排好序的结果，它们接着需要集中起来排序以确保整体排序正确。

> ## 在集群系统中深度分页
>
> 为了理解为什么深度分页是有问题的，让我们假设在一个有5个主分片的索引中搜索。当我们请求结果的第一页（结果1到10）时，每个分片产生自己最顶端10个结果然后返回它们给请求节点**(requesting node)**，它再排序这所有的50个结果以选出顶端的10个结果。
>
> 现在假设我们请求第1000页——结果10001到10010。工作方式都相同，不同的是每个分片都必须产生顶端的10010个结果。然后请求节点排序这50050个结果并丢弃50040个！
>
> 你可以看到在分布式系统中，排序结果的花费随着分页的深入而成倍增长。这也是为什么网络搜索引擎中任何语句不能返回多于1000个结果的原因。
>
> ## TIP
>
> 在《重建索引》章节我们将阐述如何能高效的检索大量文档

简易搜索

`search` API有两种表单：一种是“简易版”的查询字符串**(query string)**将所有参数通过查询字符串定义，另一种版本使用JSON完整的表示请求体**(request body)**，这种富搜索语言叫做结构化查询语句（DSL）

查询字符串搜索对于在命令行下运行点对点**(ad hoc)**查询特别有用。例如这个语句查询所有类型为 `tweet` 并在 `tweet` 字段中包含 `elasticsearch` 字符的文档：

```
GET /_all/tweet/_search?q=tweet:elasticsearch
```

下一个语句查找 `name` 字段中包含 `"john"` 和 `tweet` 字段包含 `"mary"` 的结果。实际的查询只需要：

```
+name:john +tweet:mary
```

但是百分比编码**(percent encoding)**（译者注：就是url编码）需要将查询字符串参数变得更加神秘：

```
GET /_search?q=%2Bname%3Ajohn+%2Btweet%3Amary
```

`"+"` 前缀表示语句匹配条件必须被满足。类似的 `"-"` 前缀表示条件必须不被满足。所有条件如果没有 `+` 或 `-` 表示是可选的——匹配越多，相关的文档就越多。

`_all` 字段

返回包含 `"mary"` 字符的所有文档的简单搜索：

```
GET /_search?q=mary
```

在前一个例子中，我们搜索 `tweet` 或 `name` 字段中包含某个字符的结果。然而，这个语句返回的结果在三个不同的字段中包含 `"mary"`：

- 用户的名字是“Mary”
- “Mary”发的六个推文
- 针对“@mary”的一个推文

Elasticsearch是如何设法找到三个不同字段的结果的？

当你索引一个文档，Elasticsearch把所有字符串字段值连接起来放在一个大字符串中，它被索引为一个特殊的字段 `_all`。例如，当索引这个文档：

```
{
    "tweet":    "However did I manage before Elasticsearch?",
    "date":     "2014-09-14",
    "name":     "Mary Jones",
    "user_id":  1
}
```

这好比我们增加了一个叫做 `_all` 的额外字段值：

```
"However did I manage before Elasticsearch? 2014-09-14 Mary Jones 1"
```

查询字符串在其他字段被定以前使用 `_all` 字段搜索。

> **TIP**
>
> `_all` 字段对于开始一个新应用时是一个有用的特性。之后，如果你定义字段来代替 `_all` 字段，你的搜索结果将更加可控。当 `_all` 字段不再使用，你可以停用它，这个会在《全字段》章节阐述。

更复杂的语句

下一个搜索推特的语句：

`_all` field

- `name` 字段包含 `"mary"` 或 `"john"`
- `date` 晚于 `2014-09-10`
- `_all` 字段包含 `"aggregations"` 或 `"geo"`

```
+name:(mary john) +date:>2014-09-10 +(aggregations geo)
```

编码后的查询字符串变得不太容易阅读：

```
?q=%2Bname%3A(mary+john)+%2Bdate%3A%3E2014-09-10+%2B(aggregations+geo)
```

就像你上面看到的例子，简单**(lite)**查询字符串搜索惊人的强大。它的查询语法，会在《查询字符串语法》章节阐述。参考文档允许我们简洁明快的表示复杂的查询。这对于命令行下一次性查询或者开发模式下非常有用。

然而，你可以看到简洁带来了隐晦和调试困难。而且它很脆弱——查询字符串中一个细小的语法错误，
像 `-`、`:`、`/` 或 `"` 错位就会导致返回错误而不是结果。

最后，查询字符串搜索允许任意用户在索引中任何一个字段上运行潜在的慢查询语句，可能暴露私有信息甚至使你的集群瘫痪。

> **TIP**
>
> 因为这些原因，我们不建议直接暴露查询字符串搜索给用户，除非这些用户对于你的数据和集群可信。

取而代之的，生产环境我们一般依赖全功能的请求体搜索API，它能完成前面所有的事情，甚至更多。在了解它们之前，我们首先需要看看数据是如何在Elasticsearch中被索引的。

映射**(mapping)**机制用于进行字段类型确认，将每个字段匹配为一种确定的数据类型(`string`，`number`，`booleans`，`date`等)。

分析**(analysis)**机制用于进行全文文本**(Full Text)**的分词，以建立供搜索用的反向索引。

映射及分析

当在索引中处理数据时，我们注意到一些奇怪的事。有些东西似乎被破坏了：

在索引中有12个tweets，只有一个包含日期 `2014-09-15`，但是我们看看下面查询中的 `total` hits。

```
GET /_search?q=2014              # 12 个结果
GET /_search?q=2014-09-15        # 还是 12 个结果 !
GET /_search?q=date:2014-09-15   # 1  一个结果
GET /_search?q=date:2014         # 0  个结果 !
```

为什么全日期的查询返回所有的tweets，而针对 `date` 字段进行年度查询却什么都不返回？为什么我们的结果因查询 `_all` 字段(译者注：默认所有字段中进行查询)或 `date` 字段而变得不同？

想必是因为我们的数据在 `_all` 字段的索引方式和在 `date` 字段的索引方式不同而导致。

让我们看看Elasticsearch在对 `gb` 索引中的 `tweet` 类型进行*mapping*(也称之为模式定义[注：此词有待重新定义(schema definition)])后是如何解读我们的文档结构：

```
GET /gb/_mapping/tweet
```

返回：

```
{
   "gb": {
      "mappings": {
         "tweet": {
            "properties": {
               "date": {
                  "type": "date",
                  "format": "dateOptionalTime"
               },
               "name": {
                  "type": "string"
               },
               "tweet": {
                  "type": "string"
               },
               "user_id": {
                  "type": "long"
               }
            }
         }
      }
   }
}
```

Elasticsearch为对字段类型进行猜测，动态生成了字段和类型的映射关系。返回的信息显示了 `date` 字段被识别为 `date` 类型。 `_all` 因为是默认字段所以没有在此显示，不过我们知道它是 `string` 类型。

`date` 类型的字段和 `string` 类型的字段的索引方式是不同的，因此导致查询结果的不同，这并不会让我们觉得惊讶。

你会期望每一种核心数据类型(strings, numbers, booleans及dates)以不同的方式进行索引，而这点也是现实：在Elasticsearch中他们是被区别对待的。

但是更大的区别在于确切值(exact values)(比如 `string` 类型)及全文文本(full text)之间。

这两者的区别才真的很重要 - 这是区分搜索引擎和其他数据库的根本差异。

确切值(Exact values) vs. 全文文本(Full text)

Elasticsearch中的数据可以大致分为两种类型：

确切值 及 全文文本。

确切值是确定的，正如它的名字一样。比如一个date或用户ID，也可以包含更多的字符串比如username或email地址。

确切值 `"Foo"` 和 `"foo"` 就并不相同。确切值 `2014` 和 `2014-09-15` 也不相同。

全文文本，从另一个角度来说是文本化的数据(常常以人类的语言书写)，比如一片推文(Twitter的文章)或邮件正文。

全文文本常常被称为 `非结构化数据` ，其实是一种用词不当的称谓，实际上自然语言是高度结构化的。

问题是自然语言的语法规则是如此的复杂，计算机难以正确解析。例如这个句子：

```
May is fun but June bores me.
```

到底是说的月份还是人呢？

确切值是很容易查询的，因为结果是二进制的 -- 要么匹配，要么不匹配。下面的查询很容易以SQL表达：

```
WHERE name    = "John Smith"
  AND user_id = 2
  AND date    > "2014-09-15"
```

而对于全文数据的查询来说，却有些微妙。我们不会去询问 `这篇文档是否匹配查询要求？` 。 但是，我们会询问 `这篇文档和查询的匹配程度如何？` 。换句话说，对于查询条件，这篇文档的相关性有多高？

我们很少确切的匹配整个全文文本。我们想在全文中查询包含查询文本的部分。不仅如此，我们还期望搜索引擎能理解我们的意图：

- 一个针对 `"UK"` 的查询将返回涉及 `"United Kingdom"` 的文档
- 一个针对 `"jump"` 的查询同时能够匹配 `"jumped"` ， `"jumps"` ， `"jumping"` 甚至 `"leap"`
- `"johnny walker"` 也能匹配 `"Johnnie Walker"` ， `"johnnie depp"` 及 `"Johnny Depp"`
- `"fox news hunting"` 能返回有关hunting on Fox News的故事，而 `"fox hunting news"` 也能返回关于fox hunting的新闻故事。

为了方便在全文文本字段中进行这些类型的查询，Elasticsearch首先对文本分析**(analyzes)**，然后使用结果建立一个倒排索引。我们将在以下两个章节讨论倒排索引及分析过程。

倒排索引

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。

但是在我们的倒排索引中还有些问题：

1. `"Quick"`和`"quick"`被认为是不同的单词，但是用户可能认为它们是相同的。
2. `"fox"`和`"foxes"`很相似，就像`"dog"`和`"dogs"`——它们都是同根词。
3. `"jumped"`和`"leap"`不是同根词，但意思相似——它们是同义词。

上面的索引中，搜索 `"+Quick +fox"` 不会匹配任何文档（记住，前缀 `+` 表示单词必须匹配到）。只有 `"Quick"` 和 `"fox"` 都在同一文档中才可以匹配查询，但是第一个文档包含 `"quick fox"` 且第二个文档包含 `"Quick foxes"`。（译者注：这段真啰嗦，说白了就是单复数和同义词没法匹配）

用户可以合理地希望两个文档都能匹配查询，我们也可以做得更好。

如果我们将词为统一为标准格式，这样就可以找到不是确切匹配查询，但是足以相似从而可以关联的文档。例如：

1. `"Quick"` 可以转为小写成为 `"quick"`。
2. `"foxes"` 可以被转为根形式 `"fox"`。同理 `"dogs"` 可以被转为 `"dog"`。
3. `"jumped"` 和 `"leap"` 同义就可以只索引为单个词 `"jump"`

现在的索引：

Term	Doc_1	Doc_2
brown	X	X
dog	X	X
fox	X	X
in		X
jump	X	X
lazy	X	X
over	X	X
quick	X	X
summer		X
the	X	X

但我们还未成功。我们的搜索 `"+Quick +fox"` 依旧失败，因为 `"Quick"` 的确切值已经不在索引里，不过，如果我们使用相同的标准化规则处理查询字符串的 `content` 字段，查询将变成 `"+quick +fox"`，这样就可以匹配到两个文档。

> ## IMPORTANT
>
> 这很重要。你只可以找到确实存在于索引中的词，所以索引文本和查询字符串都要标准化为相同的形式。

这个表征化和标准化的过程叫做分词**(analysis)**，这个在下节中我们讨论。

分析和分析器

分析**(analysis)**是这样一个过程：

- 首先，表征化一个文本块为适用于倒排索引单独的词**(term)**
- 然后标准化这些词为标准形式，提高它们的“可搜索性”或“查全率”

这个工作是分析器**(analyzer)**完成的。一个分析器**(analyzer)**只是一个包装用于将三个功能放到一个包里：

字符过滤器

首先字符串经过字符过滤器**(character filter)**，它们的工作是在表征化（译者注：这个词叫做断词更合适）前处理字符串。字符过滤器能够去除HTML标记，或者转换`"&"`为`"and"`。

分词器

下一步，分词器**(tokenizer)**被表征化（断词）为独立的词。一个简单的分词器**(tokenizer)**可以根据空格或逗号将单词分开（译者注：这个在中文中不适用）。

表征过滤

最后，每个词都通过所有表征过滤**(token filters)**，它可以修改词（例如将`"Quick"`转为小写），去掉词（例如停用词像`"a"`、`"and"``"the"`等等），或者增加词（例如同义词像`"jump"`和`"leap"`）

Elasticsearch提供很多开箱即用的字符过滤器，分词器和表征过滤器。这些可以组合来创建自定义的分析器以应对不同的需求。我们将在《自定义分析器》章节详细讨论。

内建的分析器

不过，Elasticsearch还附带了一些预装的分析器，你可以直接使用它们。下面我们列出了最重要的几个分析器，来演示这个字符串分词后的表现差异：

```
"Set the shape to semi-transparent by calling set_trans(5)"
```

标准分析器

标准分析器是Elasticsearch默认使用的分析器。对于文本分析，它对于任何语言都是最佳选择（译者注：就是没啥特殊需求，对于任何一个国家的语言，这个分析器就够用了）。它根据Unicode Consortium的定义的单词边界**(word boundaries)**来切分文本，然后去掉大部分标点符号。最后，把所有词转为小写。产生的结果为：

```
set, the, shape, to, semi, transparent, by, calling, set_trans, 5
```

简单分析器

简单分析器将非单个字母的文本切分，然后把每个词转为小写。产生的结果为：

```
set, the, shape, to, semi, transparent, by, calling, set, trans
```

空格分析器

空格分析器依据空格切分文本。它不转换小写。产生结果为：

```
Set, the, shape, to, semi-transparent, by, calling, set_trans(5)
```

语言分析器

特定语言分析器适用于很多语言。它们能够考虑到特定语言的特性。例如， `english` 分析器自带一套英语停用词库——像 `and` 或 `the` 这些与语义无关的通用词。这些词被移除后，因为语法规则的存在，英语单词的主体含义依旧能被理解（译者注： `stem English words` 这句不知道该如何翻译，查了字典，我理解的大概意思应该是将英语语句比作一株植物，去掉无用的枝叶，主干依旧存在，停用词好比枝叶，存在与否并不影响对这句话的理解。）。

`english` 分析器将会产生以下结果：

```
set, shape, semi, transpar, call, set_tran, 5
```

注意 `"transparent"` 、 `"calling"` 和 `"set_trans"` 是如何转为词干的。

当分析器被使用

当我们索引**(index)**一个文档，全文字段会被分析为单独的词来创建倒排索引。不过，当我们在全文字段搜索**(search)**时，我们要让查询字符串经过同样的分析流程处理，以确保这些词在索引中存在。

全文查询我们将在稍后讨论，理解每个字段是如何定义的，这样才可以让它们做正确的事：

- 当你查询全文**(full text)**字段，查询将使用相同的分析器来分析查询字符串，以产生正确的词列表。
- 当你查询一个确切值**(exact value)**字段，查询将不分析查询字符串，但是你可以自己指定。

现在你可以明白为什么《映射和分析》的开头会产生那种结果：

- `date` 字段包含一个确切值：单独的一个词 `"2014-09-15"` 。
- `_all` 字段是一个全文字段，所以分析过程将日期转为三个词： `"2014"` 、 `"09"` 和 `"15"` 。

当我们在 `_all` 字段查询 `2014` ，它一个匹配到12条推文，因为这些推文都包含词 `2014` ：

```
GET /_search?q=2014             # 12 results
```

当我们在 `_all` 字段中查询 `2014-09-15` ，首先分析查询字符串，产生匹配任一词 `2014` 、 `09` 或 `15` 的查询语句，它依旧匹配12个推文，因为它们都包含词 `2014` 。

```
GET /_search?q=2014-09-15       # 12 results !
```

当我们在 `date` 字段中查询 `2014-09-15` ，它查询一个确切的日期，然后只找到一条推文：

```
GET /_search?q=date:2014-09-15   # 1  result
```

当我们在 `date` 字段中查询 `2014` ，没有找到文档，因为没有文档包含那个确切的日期：

```
GET /_search?q=date:2014         # 0  results !
```

测试分析器

尤其当你是Elasticsearch新手时，对于如何分词以及存储到索引中理解起来比较困难。为了更好的理解如何进行，你可以使

用 `analyze` API来查看文本是如何被分析的。在查询字符串参数中指定要使用的分析器，被分析的文本做为请求体：

```
GET /_analyze?analyzer=standard
Text to analyze
```

结果中每个节点在代表一个词：

```
{
   "tokens": [
      {
         "token":        "text",
         "start_offset": 0,
         "end_offset":   4,
         "type":         "<ALPHANUM>",
         "position":     1
      },
      {
         "token":        "to",
         "start_offset": 5,
         "end_offset":   7,
         "type":         "<ALPHANUM>",
         "position":     2
      },
      {
         "token":        "analyze",
         "start_offset": 8,
         "end_offset":   15,
         "type":         "<ALPHANUM>",
         "position":     3
      }
   ]
}
```

`token` 是一个实际被存储在索引中的词。 `position` 指明词在原文本中是第几个出现的。 `start_offset` 和 `end_offset` 表示词在原文本中占据的位置。

`analyze` API 对于理解Elasticsearch索引的内在细节是个非常有用的工具，随着内容的推进，我们将继续讨论它。

指定分析器

当Elasticsearch在你的文档中探测到一个新的字符串字段，它将自动设置它为全文 `string` 字段并用 `standard` 分析器分析。

你不可能总是想要这样做。也许你想使用一个更适合这个数据的语言分析器。或者，你只想把字符串字段当作一个普通的字段——不做任何分析，只存储确切值，就像字符串类型的用户ID或者内部状态字段或者标签。

为了达到这种效果，我们必须通过映射**(mapping)**人工设置这些字段。

映射

正如《数据吞吐》一节所说，索引中每个文档都有一个类型**(type)**。 每个类型拥有自己的映射**(mapping)**或者模式定义**(schema definition)**。一个映射定义了字段类型，每个字段的数据类型，以及字段被Elasticsearch处理的方式。映射还用于设置关联到类型上的元数据。

在《映射》章节我们将探讨映射的细节。这节我们只是带你入门。

核心简单字段类型

Elasticsearch支持以下简单字段类型：

类型	表示的数据类型
String	`string`
Whole number	`byte` , `short` , `integer` , `long`
Floating point	`float` , `double`
Boolean	`boolean`
Date	`date`

当你索引一个包含新字段的文档——一个之前没有的字段——Elasticsearch将使用动态映射猜测字段类型，这类型来自于JSON的基本数据类型，使用以下规则：

JSON type	Field type
Boolean: `true` or `false`	`"boolean"`
Whole number: `123`	`"long"`
Floating point: `123.45`	`"double"`
String, valid date: `"2014-09-15"`	`"date"`
String: `"foo bar"`	`"string"`

> ### 注意
>
> 这意味着，如果你索引一个带引号的数字——`"123"`，它将被映射为`"string"`类型，而不是`"long"`类型。然而，如果字段已经被映射为`"long"`类型，Elasticsearch将尝试转换字符串为long，并在转换失败时会抛出异常。

查看映射

我们可以使用`_mapping`后缀来查看Elasticsearch中的映射。在本章开始我们已经找到索引`gb`类型`tweet`中的映射：

```
GET /gb/_mapping/tweet
```

这展示给了我们字段的映射（叫做属性**(properties)**），这些映射是Elasticsearch在创建索引时动态生成的：

```
{
   "gb": {
      "mappings": {
         "tweet": {
            "properties": {
               "date": {
                  "type": "date",
                  "format": "dateOptionalTime"
               },
```

```
                "name": {
                   "type": "string"
                },
                "tweet": {
                   "type": "string"
                },
                "user_id": {
                   "type": "long"
                }
             }
          }
       }
    }
}
```

> 小提示
>
> 错误的映射，例如把 `age` 字段映射为 `string` 类型而不是 `integer` 类型，会造成查询结果混乱。
>
> 要检查映射类型，而不是假设它是正确的！

自定义字段映射

映射中最重要的字段参数是 `type` 。除了 `string` 类型的字段，你可能很少需要映射其他的 `type` ：

```
{
    "number_of_clicks": {
        "type": "integer"
    }
}
```

`string` 类型的字段，默认的，考虑到包含全文本，它们的值在索引前要经过分析器分析，并且在全文搜索此字段前要把查询语句做分析处理。

对于 `string` 字段，两个最重要的映射参数是 `index` 和 `analyer` 。

index

`index` 参数控制字符串以何种方式被索引。它包含以下三个值当中的一个：

值	解释
`analyzed`	首先分析这个字符串，然后索引。换言之，以全文形式索引此字段。
`not_analyzed`	索引这个字段，使之可以被搜索，但是索引内容和指定值一样。不分析此字段。
`no`	不索引这个字段。这个字段不能为搜索到。

`string` 类型字段默认值是 `analyzed` 。如果我们想映射字段为确切值，我们需要设置它为 `not_analyzed` ：

```
{
    "tag": {
        "type":     "string",
        "index":    "not_analyzed"
    }
}
```

> 其他简单类型—— `long` 、 `double` 、 `date` 等等——也接受 `index` 参数，但相应的值只能是 `no` 和 `not_analyzed` ，它们的值不能被分析。

分析

对于 `analyzed` 类型的字符串字段，使用 `analyzer` 参数来指定哪一种分析器将在搜索和索引的时候使用。默认的，Elasticsearch使用 `standard` 分析器，但是你可以通过指定一个内建的分析器来更改它，例如 `whitespace`、 `simple` 或 `english`。

```
{
    "tweet": {
        "type":     "string",
        "analyzer": "english"
    }
}
```

在《自定义分析器》章节我们将告诉你如何定义和使用自定义的分析器。

更新映射

你可以在第一次创建索引的时候指定映射的类型。此外，你也可以晚些时候为新类型添加映射（或者为已有的类型更新映射）。

> ### 重要
>
> 你可以向已有映射中增加字段，但你不能修改它。如果一个字段在映射中已经存在，这可能意味着那个字段的数据已经被索引。如果你改变了字段映射，那已经被索引的数据将错误并且不能被正确的搜索到。

我们可以更新一个映射来增加一个新字段，但是不能把已有字段的类型那个从 `analyzed` 改到 `not_analyzed`。

为了演示两个指定的映射方法，让我们首先删除索引 `gb`：

```
DELETE /gb
```

然后创建一个新索引，指定 `tweet` 字段的分析器为 `english`：

```
PUT /gb <1>
{
  "mappings": {
    "tweet" : {
      "properties" : {
        "tweet" : {
          "type" :    "string",
          "analyzer": "english"
        },
        "date" : {
          "type" :   "date"
        },
        "name" : {
          "type" :   "string"
        },
        "user_id" : {
          "type" :   "long"
        }
      }
    }
  }
}
```

`<1>` 这将创建包含 `mappings` 的索引，映射在请求体中指定。

再后来，我们决定在 `tweet` 的映射中增加一个新的 `not_analyzed` 类型的文本字段，叫做 `tag`，使用 `_mapping` 后缀：

```
PUT /gb/_mapping/tweet
{
  "properties" : {
    "tag" : {
```

```
        "type" :    "string",
        "index":    "not_analyzed"
      }
    }
  }
```

注意到我们不再需要列出所有的已经存在的字段，因为我们没法修改他们。我们的新字段已经被合并至存在的那个映射中。

测试映射

你可以通过名字使用 `analyze` API测试字符串字段的映射。对比这两个请求的输出：

```
GET /gb/_analyze?field=tweet
Black-cats <1>

GET /gb/_analyze?field=tag
Black-cats <1>
```

`<1>` 我们想要分析的文本被放在请求体中。

`tweet` 字段产生两个词， `"black"` 和 `"cat"` , `tag` 字段产生单独的一个词 `"Black-cats"` 。换言之，我们的映射工作正常。

复合核心字段类型

除了之前提到的简单的标量类型，JSON还有 `null` 值，数组和对象，所有这些Elasticsearch都支持：

多值字段

我们想让 `tag` 字段包含多个字段，这非常有可能发生。我们可以索引一个标签数组来代替单一字符串：

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

对于数组不需要特殊的映射。任何一个字段可以包含零个、一个或多个值，同样对于全文字段将被分析并产生多个词。

言外之意，这意味着数组中所有值必须为同一类型。你不能把日期和字符窜混合。如果你创建一个新字段，这个字段索引了一个数组，Elasticsearch将使用第一个值的类型来确定这个新字段的类型。

当你从Elasticsearch中取回一个文档，任何一个数组的顺序和你索引它们的顺序一致。你取回的 `_source` 字段的顺序同样与索引它们的顺序相同。

然而，数组是做为多值字段被索引的，它们没有顺序。在搜索阶段你不能指定“第一个值”或者“最后一个值”。倒不如把数组当作一个值集合**(gag of values)**

==== Empty fields

空字段

当然数组可以是空的。这等价于有零个值。事实上，Lucene没法存放 `null` 值，所以一个 `null` 值的字段被认为是空字段。

这四个字段将被识别为空字段而不被索引：

```
"empty_string":             "",
"null_value":               null,
"empty_array":              [],
"array_with_null_value":    [ null ]
```

多层对象

我们需要讨论的最后一个自然JSON数据类型是对象**(object)**——在其它语言中叫做hashed、hashmaps、dictionaries 或者 associative arrays.

内部对象**(inner objects)**经常用于嵌入一个实体或对象里的另一个地方。例如，做在 `tweet` 文档中 `user_name` 和 `user_id` 的替代，我们可以这样写：

```
{
    "tweet":            "Elasticsearch is very flexible",
    "user": {
        "id":           "@johnsmith",
        "gender":       "male",
        "age":          26,
        "name": {
            "full":     "John Smith",
            "first":    "John",
            "last":     "Smith"
        }
    }
}
```

内部对象的映射

Elasticsearch 会动态的检测新对象的字段，并且映射它们为 `object` 类型，将每个字段加到 `properties` 字段下

```
{
  "gb": {
    "tweet": { <1>
      "properties": {
        "tweet":            { "type": "string" },
        "user": { <2>
          "type":             "object",
          "properties": {
            "id":           { "type": "string" },
            "gender":       { "type": "string" },
            "age":          { "type": "long"   },
            "name":   { <2>
              "type":         "object",
              "properties": {
                "full":     { "type": "string" },
                "first":    { "type": "string" },
                "last":     { "type": "string" }
              }
            }
          }
        }
      }
    }
  }
}
```

<1> 根对象.

<2> 内部对象.

The mapping for the `user` and `name` fields have a similar structure to the mapping for the `tweet` type itself. In fact, the `type` mapping is just a special type of `object` mapping, which we refer to as the *root object*. It is just the same as any other object, except that it has some special top-level fields for document metadata, like `_source`, the `_all` field etc.

对 `user` 和 `name` 字段的映射与 `tweet` 类型自己很相似。事实上， `type` 映射只是 `object` 映射的一种特殊类型，我们将 `object` 称为根对象。它与其他对象一模一样，除非它有一些特殊的顶层字段，比如 `_source` ， `_all` 等等。

内部对象是怎样被索引的

Lucene doesn't understand inner objects. A Lucene document consists of a flat list of key-value pairs. In order for Elasticsearch to index inner objects usefully, it converts our document into something like this:

```
{
    "tweet":            [elasticsearch, flexible, very],
    "user.id":          [@johnsmith],
    "user.gender":      [male],
    "user.age":         [26],
    "user.name.full":   [john, smith],
    "user.name.first":  [john],
    "user.name.last":   [smith]
}
```

Inner fields can be referred to by name, eg `"first"`. To distinguish between two fields that have the same name we can use the full *path*, eg `"user.name.first"` or even the `type` name plus the path: `"tweet.user.name.first"`.

NOTE: In the simple flattened document above, there is no field called `user` and no field called `user.name`. Lucene only indexes scalar or simple values, not complex datastructures.

[[object-arrays]] ==== Arrays of inner objects

Finally, consider how an array containing inner objects would be indexed. Let's say we have a `followers` array which looks

like this:

[source,js]

{ "followers": [{ "age": 35, "name": "Mary White"}, { "age": 26, "name": "Alex Jones"}, { "age": 19, "name": "Lisa Smith"}]

}

This document will be flattened as we described above, but the result will look like this:

[source,js]

{ "followers.age": [19, 26, 35], "followers.name": [alex, jones, lisa, smith, mary, white]

}

The correlation between `{age: 35}` and `{name: Mary White}` has been lost as each multi-value field is just a bag of values, not an ordered array. This is sufficient for us to ask:

- *Is there a follower who is 26 years old?*

but we can't get an accurate answer to:

- *Is there a follower who is 26 years old* ***and who is called Alex Jones?***

Correlated inner objects, which are able to answer queries like these, are called *nested* objects, and we will discuss them later on in <>.

请求体查询

简单查询语句(lite)是一种有效的命令行*adhoc*查询。但是，如果你想要善用搜索，你必须使用请求体查询(request body `search`)API。之所以这么称呼，是因为大多数的参数以JSON格式所容纳而非查询字符串。

请求体查询(下文简称查询)，并不仅仅用来处理查询，而且还可以高亮返回结果中的片段，并且给出帮助你的用户找寻最好结果的相关数据建议。

空查询

我们以最简单的 `search` API开始，空查询将会返回索引中所有的文档。

```
GET /_search
{} <1>
```

- <1> 这是一个空查询数据。

同字符串查询一样，你可以查询一个，多个或 `_all` 索引(indices)或类型(types)：

```
GET /index_2014*/type1,type2/_search
{}
```

你可以使用 `from` 及 `size` 参数进行分页：

```
GET /_search
{
  "from": 30,
  "size": 10
}
```

携带内容的 `GET` 请求？

任何一种语言(特别是js)的HTTP库都不允许 `GET` 请求中携带交互数据。 事实上，有些用户很惊讶 `GET` 请求中居然会允许携带交互数据。

真实情况是，http://tools.ietf.org/html/rfc7231#page-24[RFC 7231]， 一份规定HTTP语义及内容的RFC中并未规定 `GET` 请求中允许携带交互数据！ 所以，有些HTTP服务允许这种行为，而另一些(特别是缓存代理)，则不允许这种行为。

Elasticsearch的作者们倾向于使用 `GET` 提交查询请求，因为他们觉得这个词相比 `POST` 来说，能更好的描述这种行为。 然而，因为携带交互数据的 `GET` 请求并不被广泛支持，所以 `search` API同样支持 `POST` 请求，类似于这样：

```
POST /_search
{
  "from": 30,
  "size": 10
}
```

这个原理同样应用于其他携带交互数据的 `GET` API请求中。

我们将在后续的章节中讨论聚合查询，但是现在我们把关注点仅放在查询语义上。

相对于神秘的查询字符串方法，请求体查询允许我们使用结构化查询Query DSL(Query Domain Specific Language)

结构化查询 Query DSL

结构化查询是一种灵活的，多表现形式的查询语言。 Elasticsearch在一个简单的JSON接口中用结构化查询来展现Lucene绝大多数能力。 你应当在你的产品中采用这种方式进行查询。它使得你的查询更加灵活，精准，易于阅读并且易于debug。

使用结构化查询，你需要传递 `query` 参数：

```
GET /_search
{
    "query": YOUR_QUERY_HERE
}
```

空查询 - `{}` - 在功能上等同于使用 `match_all` 查询子句，正如其名字一样，匹配所有的文档：

```
GET /_search
{
    "query": {
        "match_all": {}
    }
}
```

查询子句

一个查询子句一般使用这种结构：

```
{
    QUERY_NAME: {
        ARGUMENT: VALUE,
        ARGUMENT: VALUE,...
    }
}
```

或指向一个指定的字段：

```
{
    QUERY_NAME: {
        FIELD_NAME: {
            ARGUMENT: VALUE,
            ARGUMENT: VALUE,...
        }
    }
}
```

例如，你可以使用 `match` 查询子句用来找寻在 `tweet` 字段中找寻包含 `elasticsearch` 的成员：

```
{
    "match": {
        "tweet": "elasticsearch"
    }
}
```

完整的查询请求会是这样：

```
GET /_search
{
    "query": {
        "match": {
```

```
            "tweet": "elasticsearch"
        }
    }
}
```

合并多子句

查询子句就像是搭积木一样，可以合并简单的子句为一个复杂的查询语句，比如：

- 简单子句(*leaf clauses*)(比如 `match` 子句)用以在将查询字符串与一个字段(或多字段)进行比较
- 复合子句(*compound*)用以合并其他的子句。例如， `bool` 子句允许你合并其他的合法子句，无论是 `must` ， `must_not` 还是 `should` ：

```
{
    "bool": {
        "must":     { "match": { "tweet": "elasticsearch" }},
        "must_not": { "match": { "name":  "mary" }},
        "should":   { "match": { "tweet": "full text" }}
    }
}
```

复合子句能合并 任意其他查询子句，包括其他的复合子句。 这就意味着复合子句可以相互嵌套，从而实现非常复杂的逻辑。

以下实例查询在inbox中或未标记spam的邮件中找出包含 `"business opportunity"` 的星标(starred)邮件：

```
{
    "bool": {
        "must": { "match":      { "email": "business opportunity" }},
        "should": [
             { "match":         { "starred": true }},
             { "bool": {
                   "must":      { "folder": "inbox" }},
                   "must_not":  { "spam": true }}
             }}
        ],
        "minimum_should_match": 1
    }
}
```

不用担心这个例子的细节，我们将在后面详细解释它。 重点是复合子句可以合并多种子句为一个单一的查询，无论是简单子句还是其他的复合子句。

translate by williamzhao

查询与过滤

前面我们讲到的是关于结构化查询语句，事实上我们可以使用两种结构化语句： 结构化查询（Query DSL）和结构化过滤（Filter DSL）。 查询与过滤语句非常相似，但是它们由于使用目的不同而稍有差异。

一条过滤语句会询问每个文档的字段值是否包含着特定值：

- 是否 `created` 的日期范围在 `2013` 到 `2014` ？
- 是否 `status` 字段中包含单词 "published" ？
- 是否 `lat_lon` 字段中的地理位置与目标点相距不超过10km ？

一条查询语句与过滤语句相似，但问法不同：

查询语句会询问每个文档的字段值与特定值的匹配程度如何？

查询语句的典型用法是为了找到文档：

- 查找与 `full text search` 这个词语最佳匹配的文档
- 查找包含单词 `run` ，但是也包含 `runs` , `running` , `jog` 或 `sprint` 的文档
- 同时包含着 `quick` , `brown` 和 `fox` --- 单词间离得越近，该文档的相关性越高
- 标识着 `lucene` , `search` 或 `java` --- 标识词越多，该文档的相关性越高

一条查询语句会计算每个文档与查询语句的相关性，会给出一个相关性评分 `_score` ，并且 按照相关性对匹配到的文档进行排序。 这种评分方式非常适用于一个没有完全配置结果的全文本搜索。

性能差异

使用过滤语句得到的结果集 -- 一个简单的文档列表，快速匹配运算并存入内存是十分方便的， 每个文档仅需要1个字节。这些缓存的过滤结果集与后续请求的结合使用是非常高效的。

查询语句不仅要查找相匹配的文档，还需要计算每个文档的相关性，所以一般来说查询语句要比 过滤语句更耗时，并且查询结果也不可缓存。

幸亏有了倒排索引，一个只匹配少量文档的简单查询语句在百万级文档中的查询效率会与一条经过缓存 的过滤语句旗鼓相当，甚至略占上风。 但是一般情况下，一条经过缓存的过滤查询要远胜一条查询语句的执行效率。

过滤语句的目的就是缩小匹配的文档结果集，所以需要仔细检查过滤条件。

什么情况下使用

原则上来说，使用查询语句做全文本搜索或其他需要进行相关性评分的时候，剩下的全部用过滤语句

最重要的查询过滤语句

Elasticsearch 提供了丰富的查询过滤语句，而有一些是我们较常用到的。 我们将会在后续的《深入搜索》中展开讨论，现在我们快速的介绍一下 这些最常用到的查询过滤语句。

`term` 过滤

`term` 主要用于精确匹配哪些值，比如数字，日期，布尔值或 `not_analyzed` 的字符串(未经分析的文本数据类型)：

```
    { "term": { "age":    26           }}
    { "term": { "date":   "2014-09-01" }}
    { "term": { "public": true         }}
    { "term": { "tag":    "full_text"  }}
```

`terms` 过滤

`terms` 跟 `term` 有点类似，但 `terms` 允许指定多个匹配条件。 如果某个字段指定了多个值，那么文档需要一起去做匹配：

```
{
    "terms": {
        "tag": [ "search", "full_text", "nosql" ]
        }
}
```

`range` 过滤

`range` 过滤允许我们按照指定范围查找一批数据：

```
{
    "range": {
        "age": {
            "gte":  20,
            "lt":   30
        }
    }
}
```

范围操作符包含：

`gt` :: 大于

`gte` :: 大于等于

`lt` :: 小于

`lte` :: 小于等于

`exists` 和 `missing` 过滤

`exists` 和 `missing` 过滤可以用于查找文档中是否包含指定字段或没有某个字段， 类似于SQL语句中的 `IS_NULL` 条件

```
{
    "exists":   {
```

```
        "field":    "title"
    }
}
```

这两个过滤只是针对已经查出一批数据来，但是想区分出某个字段是否存在的时候使用。

`bool` 过滤

`bool` 过滤可以用来合并多个过滤条件查询结果的布尔逻辑，它包含一下操作符：

`must` :: 多个查询条件的完全匹配,相当于 `and` 。

`must_not` :: 多个查询条件的相反匹配，相当于 `not` 。

`should` :: 至少有一个查询条件匹配, 相当于 `or` 。

这些参数可以分别继承一个过滤条件或者一个过滤条件的数组：

```
{
    "bool": {
        "must":     { "term": { "folder": "inbox" }},
        "must_not": { "term": { "tag":    "spam"  }},
        "should": [
                    { "term": { "starred": true   }},
                    { "term": { "unread":  true   }}
        ]
    }
}
```

`match_all` 查询

使用 `match_all` 可以查询到所有文档，是没有查询条件下的默认语句。

```
{
    "match_all": {}
}
```

此查询常用于合并过滤条件。 比如说你需要检索所有的邮箱,所有的文档相关性都是相同的，所以得到的 `_score` 为1

`match` 查询

`match` 查询是一个标准查询，不管你需要全文本查询还是精确查询基本上都要用到它。

如果你使用 `match` 查询一个全文本字段，它会在真正查询之前用分析器先分析 `match` 一下查询字符：

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

如果用 `match` 下指定了一个确切值，在遇到数字，日期，布尔值或者 `not_analyzed` 的字符串时，它将为你搜索你给定的值：

```
{ "match": { "age":    26           }}
{ "match": { "date":   "2014-09-01" }}
{ "match": { "public": true         }}
{ "match": { "tag":    "full_text"  }}
```

> 提示： 做精确匹配搜索时，你最好用过滤语句，因为过滤语句可以缓存数据。

不像我们在《简单搜索》中介绍的字符查询， `match` 查询不可以用类似"+usid:2 +tweet:search"这样的语句。 它只能就指定某个确切字段某个确切的值进行搜索，而你要做的就是为它指定正确的字段名以避免语法错误。

`multi_match` 查询

`multi_match` 查询允许你做 `match` 查询的基础上同时搜索多个字段：

```
{
    "multi_match": {
        "query":    "full text search",
        "fields":   [ "title", "body" ]
    }
}
```

`bool` 查询

`bool` 查询与 `bool` 过滤相似，用于合并多个查询子句。不同的是， `bool` 过滤可以直接给出是否匹配成功， 而 `bool` 查询要计算每一个查询子句的 `_score` （相关性分值）。

`must` :: 查询指定文档一定要被包含。

`must_not` :: 查询指定文档一定不要被包含。

`should` :: 查询指定文档，有则可以为文档相关性加分。

以下查询将会找到 `title` 字段中包含 "how to make millions"，并且 "tag" 字段没有被标为 `spam` 。 如果有标识为 "starred" 或者发布日期为2014年之前，那么这些匹配的文档将比同类网站等级高：

```
{
    "bool": {
        "must":     { "match": { "title": "how to make millions" }},
        "must_not": { "match": { "tag":   "spam" }},
        "should": [
            { "match": { "tag": "starred" }},
            { "range": { "date": { "gte": "2014-01-01" }}}
        ]
    }
}
```

> 提示： 如果 `bool` 查询下没有 `must` 子句，那至少应该有一个 `should` 子句。但是 如果有 `must` 子句，那么没有 `should` 子句也可以进行查询。

查询与过滤条件的合并

查询语句和过滤语句可以放在各自的上下文中。 在 ElasticSearch API 中我们会看到许多带有 `query` 或 `filter` 的语句。 这些语句既可以包含单条 query 语句，也可以包含一条 filter 子句。 换句话说，这些语句需要首先创建一个 `query` 或 `filter` 的上下文关系。

复合查询语句可以加入其他查询子句，复合过滤语句也可以加入其他过滤子句。 通常情况下，一条查询语句需要过滤语句的辅助，全文本搜索除外。

所以说，查询语句可以包含过滤子句，反之亦然。 以便于我们切换 query 或 filter 的上下文。这就要求我们在读懂需求的同时构造正确有效的语句。

带过滤的查询语句

过滤一条查询语句

比如说我们有这样一条查询语句:

```
{
    "match": {
        "email": "business opportunity"
    }
}
```

然后我们想要让这条语句加入 `term` 过滤，在收信箱中匹配邮件：

```
{
    "term": {
        "folder": "inbox"
    }
}
```

`search` API中只能包含 `query` 语句，所以我们需要用 `filtered` 来同时包含 "query" 和 "filter" 子句：

```
{
    "filtered": {
        "query":  { "match": { "email": "business opportunity" }},
        "filter": { "term":  { "folder": "inbox" }}
    }
}
```

我们在外层再加入 `query` 的上下文关系：

```
GET /_search
{
    "query": {
        "filtered": {
            "query":  { "match": { "email": "business opportunity" }},
            "filter": { "term": { "folder": "inbox" }}
        }
    }
}
```

单条过滤语句

在 `query` 上下文中，如果你只需要一条过滤语句，比如在匹配全部邮件的时候，你可以 省略 `query` 子句：

```
GET /_search
{
    "query": {
        "filtered": {
            "filter":   { "term": { "folder": "inbox" }}
        }
    }
}
```

如果一条查询语句没有指定查询范围，那么它默认使用 `match_all` 查询，所以上面语句 的完整形式如下：

```
GET /_search
{
    "query": {
        "filtered": {
            "query":    { "match_all": {}},
            "filter":   { "term": { "folder": "inbox" }}
        }
    }
}
```

查询语句中的过滤

有时候，你需要在 filter 的上下文中使用一个 query 子句。下面的语句就是一条带有查询功能 的过滤语句， 这条语句可以过滤掉看起来像垃圾邮件的文档：

```
GET /_search
{
    "query": {
        "filtered": {
            "filter":   {
                "bool": {
                    "must":     { "term":  { "folder": "inbox" }},
                    "must_not": {
                        "query": { <1>
                            "match": { "email": "urgent business proposal" }
                        }
                    }
                }
            }
        }
    }
}
```

<1> 过滤语句中可以使用 `query` 查询的方式代替 `bool` 过滤子句。

> 提示： 我们很少用到的过滤语句中包含查询，保留这种用法只是为了语法的完整性。 只有在过滤中用到全文本匹配的时候才会使用这种结构。

验证查询

查询语句可以变得非常复杂，特别是与不同的分析器和字段映射相结合后，就会有些难度。

`validate` API 可以验证一条查询语句是否合法。

```
GET /gb/tweet/_validate/query
{
   "query": {
      "tweet" : {
         "match" : "really powerful"
      }
   }
}
```

以上请求的返回值告诉我们这条语句是非法的：

```
{
  "valid" :         false,
  "_shards" : {
    "total" :       1,
    "successful" :  1,
    "failed" :      0
  }
}
```

理解错误信息

想知道语句非法的具体错误信息，需要加上 `explain` 参数：

```
GET /gb/tweet/_validate/query?explain <1>
{
   "query": {
      "tweet" : {
         "match" : "really powerful"
      }
   }
}
```

<1> `explain` 参数可以提供语句错误的更多详情。

很显然，我们把 query 语句的 `match` 与字段名位置弄反了：

```
{
  "valid" :     false,
  "_shards" :   { ... },
  "explanations" : [ {
    "index" :   "gb",
    "valid" :   false,
    "error" :   "org.elasticsearch.index.query.QueryParsingException:
                 [gb] No query registered for [tweet]"
  } ]
}
```

理解查询语句

如果是合法语句的话，使用 `explain` 参数可以返回一个带有查询语句的可阅读描述， 可以帮助了解查询语句在ES中是如何执行的：

```
GET /_validate/query?explain
{
   "query": {
      "match" : {
         "tweet" : "really powerful"
      }
   }
}
```

`explanation` 会为每一个索引返回一段描述，因为每个索引会有不同的映射关系和分析器：

```
{
  "valid" :         true,
  "_shards" :       { ... },
  "explanations" : [ {
    "index" :       "us",
    "valid" :       true,
    "explanation" : "tweet:really tweet:powerful"
  }, {
    "index" :       "gb",
    "valid" :       true,
    "explanation" : "tweet:really tweet:power"
  } ]
}
```

从返回的 `explanation` 你会看到 `match` 是如何为查询字符串 `"really powerful"` 进行查询的， 首先，它被拆分成两个独立的词分别在 `tweet` 字段中进行查询。

而且，在索引 `us` 中这两个词为 `"really"` 和 `"powerful"`，在索引 `gb` 中被拆分成 `"really"` 和 `"power"`。 这是因为我们在索引 `gb` 中使用了 `english` 分析器。

结语

这一章详细介绍了如何在项目中使用常见的查询语句。

也就是说，想要完全掌握搜索和结构化查询，还需要在工作中花费大量的时间来理解ES的工作方式。

更高级的部分，我们将会在《深入搜索》中详细讲解，但是在讲解之前，你还需要理解查询结果是如何进行排序的，

下一章我们将学习如何根据相关性对查询结果进行排序以及指定排序过程。

相关性排序

默认情况下，结果集会按照相关性进行排序 -- 相关性越高，排名越靠前。 这一章我们会讲述相关性是什么以及它是如何计算的。 在此之前，我们先看一下 `sort` 参数的使用方法。

排序方式

为了使结果可以按照相关性进行排序，我们需要一个相关性的值。在ElasticSearch的查询结果中， 相关性分值会用 `_score` 字段来给出一个浮点型的数值，所以默认情况下，结果集以 `_score` 进行倒序排列。

有时，即便如此，你还是没有一个有意义的相关性分值。比如，以下语句返回所有tweets中 `user_id` 是否 包含值 `1`：

```
GET /_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "term" : {
                    "user_id" : 1
                }
            }
        }
    }
}
```

过滤语句与 `_score` 没有关系，但是有隐含的查询条件 `match_all` 为所有的文档的 `_score` 设值为 `1`。 也就相当于所有的文档相关性是相同的。

字段值排序

下面例子中，对结果集按照时间排序，这也是最常见的情形，将最新的文档排列靠前。 我们使用 `sort` 参数进行排序：

```
GET /_search
{
    "query" : {
        "filtered" : {
            "filter" : { "term" : { "user_id" : 1 }}
        }
    },
    "sort": { "date": { "order": "desc" }}
}
```

你会发现这里有两个不同点：

```
"hits" : {
    "total" :           6,
    "max_score" :       null, <1>
    "hits" : [ {
        "_index" :      "us",
        "_type" :       "tweet",
        "_id" :         "14",
        "_score" :      null, <1>
        "_source" :     {
             "date":    "2014-09-24",
             ...
        },
        "sort" :        [ 1411516800000 ] <2>
    },
    ...
}
```

<1> `_score` 字段没有经过计算，因为它没有用作排序。

<2> `date` 字段被转为毫秒当作排序依据。

首先，在每个结果中增加了一个 `sort` 字段，它所包含的值是用来排序的。 在这个例子当中 `date` 字段在内部被转为毫秒，即长整型数字 `1411516800000` 等同于日期字符串 `2014-09-24 00:00:00 UTC`。

其次就是 `_score` 和 `max_score` 字段都为 `null`。计算 `_score` 是比较消耗性能的, 而且通常主要用作排序 -- 我们不是用相关性进行排序的时候，就不需要统计其相关性。 如果你想强制计算其相关性，可以设置 `track_scores` 为 `true`。

默认排序

作为缩写，你可以只指定要排序的字段名称：

```
    "sort": "number_of_children"
```

字段值默认以顺序排列，而 `_score` 默认以倒序排列。

多级排序

如果我们想要合并一个查询语句，并且展示所有匹配的结果集使用第一排序是 `date`，第二排序是 `_score`：

```
GET /_search
{
    "query" : {
        "filtered" : {
            "query":   { "match": { "tweet": "manage text search" }},
            "filter" : { "term" : { "user_id" : 2 }}
        }
    },
    "sort": [
        { "date":   { "order": "desc" }},
        { "_score": { "order": "desc" }}
    ]
}
```

排序是很重要的。结果集会先用第一排序字段来排序，当用用作第一字段排序的值相同的时候， 然后再用第二字段对第一排序值相同的文档进行排序，以此类推。

多级排序不需要包含 `_score` -- 你可以使用几个不同的字段，如位置距离或者自定义数值。

字符串参数排序

字符查询也支持自定义排序，在查询字符串使用 `sort` 参数就可以：

```
GET /_search?sort=date:desc&sort=_score&q=search
```

为多值字段排序

在为一个字段的多个值进行排序的时候， 其实这些值本来是没有固定的排序的-- 一个拥有多值的字段就是一个集合， 你准备

以哪一个作为排序依据呢？

对于数字和日期，你可以从多个值中取出一个来进行排序，你可以使用 `min`，`max`，`avg` 或 `sum` 这些模式。 比说你可以在 `dates` 字段中用最早的日期来进行排序：

```
"sort": {
    "dates": {
        "order": "asc",
        "mode":  "min"
    }
}
```

多值字段字符串排序

`analyzed` 字符串字段同时也是多值字段，在这些字段上排序往往得不到你想要的值。 比如你分析一个字符 `"fine old art"`，它最终会得到三个值。例如我们想要按照第一个词首字母排序， 如果第一个单词相同的话，再用第二个词的首字母排序，以此类推，可惜 ElasticSearch 在进行排序时 是得不到这些信息的。

当然你可以使用 `min` 和 `max` 模式来排（默认使用的是 `min` 模式）但它是依据 `art` 或者 `old` 排序， 而不是我们所期望的那样。

为了使一个string字段可以进行排序，它必须只包含一个词：即完整的 `not_analyzed` 字符串。 当然我们需要对字段进行全文本搜索的时候还必须使用 `analyzed`。

在 `_source` 下相同的字符串上排序两次会造成不必要的资源浪费。 而我们想要的是一个字段中同时包含这两种索引方式。现在我们介绍一个在所有核心字段类型上通用的参数 `fields`， 这样我们就可以改变它的mapping：

```
"tweet": {
    "type":     "string",
    "analyzer": "english"
}
```

改变后的多值字段mapping如下：

```
"tweet": { <1>
    "type":     "string",
    "analyzer": "english",
    "fields": {
        "raw": { <2>
            "type":  "string",
            "index": "not_analyzed"
        }
    }
}
```

<1> `tweet` 字段用于全文本的 `analyzed` 索引方式不变。

<2> 新增的 `tweet.raw` 子字段索引方式是 `not_analyzed`。

现在，在给数据重建索引后，我们既可以使用 `tweet` 字段进行全文本搜索，也可以用 `tweet.raw` 字段进行排序：

```
GET /_search
{
    "query": {
        "match": {
            "tweet": "elasticsearch"
        }
    },
    "sort": "tweet.raw"
}
```

> 警告： 对 `analyzed` 字段进行强制排序会消耗大量内存。 详情请查阅《字段类型简介》相关内容。

相关性简介

我们曾经讲过，默认情况下，返回结果是按相关性倒序排列的。 但是什么是相关性？ 相关性如何计算？

每个文档都有相关性评分，用一个相对的浮点数字段 `_score` 来表示 -- `_score` 的评分越高，相关性越高。

查询语句会为每个文档添加一个 `_score` 字段。评分的计算方式取决于不同的查询类型 -- 不同的查询语句用于不同的目的： `fuzzy` 查询会计算与关键词的拼写相似程度， `terms` 查询会计算 找到的内容与关键词组成部分匹配的百分比，但是一般意义上我们说的全文本搜索是指计算内容与关键词的类似程度。

ElasticSearch的相似度算法被定义为 TF/IDF，即检索词频率/反向文档频率，包括一下内容：

检索词频率::

检索词在该字段出现的频率？出现频率越高，相关性也越高。 字段中出现过5次要比只出现过1次的相关性高。

反向文档频率::

每个检索词在索引中出现的频率？频率越高，相关性越低。 检索词出现在多数文档中会比出现在少数文档中的权重更低， 即检验一个检索词在文档中的普遍重要性。

字段长度准则::

字段的长度是多少？长度越长，相关性越低。 检索词出现在一个短的 `title` 要比同样的词出现在一个长的 `content` 字段。

单个查询可以使用TF/IDF评分标准或其他方式，比如短语查询中检索词的距离或模糊查询里的检索词相似度。

相关性并不只是全文本检索的专利。也适用于 `yes|no` 的子句，匹配的子句越多，相关性评分越高。

如果多条查询子句被合并为一条复合查询语句，比如 `bool` 查询，则每个查询子句计算得出的评分会被合并到总的相关性评分中。

理解评分标准

当调试一条复杂的查询语句时，想要理解相关性评分 `_score` 是比较困难的。ElasticSearch 在 每个查询语句中都有一个explain参数，将 `explain` 设为 `true` 就可以得到更详细的信息。

```
GET /_search?explain <1>
{
   "query"   : { "match" : { "tweet" : "honeymoon" }}
}
```

<1> `explain` 参数可以让返回结果添加一个 `_score` 评分的得来依据。

增加一个 `explain` 参数会为每个匹配到的文档产生一大堆额外内容，但是花时间去理解它是很有意义的。 如果现在看不明白也没关系 -- 等你需要的时候再来回顾这一节就行。下面我们来一点点的了解这块知识点。

首先，我们看一下普通查询返回的元数据：

```
{
    "_index" :      "us",
    "_type" :       "tweet",
    "_id" :         "12",
    "_score" :      0.076713204,
    "_source" :     { ... trimmed ... },
```

```
}
```

这里加入了该文档来自于哪个节点哪个分片上的信息，这对我们是比较有帮助的，因为词频率和 文档频率是在每个分片中计算出来的，而不是每个索引中：

```
    "_shard" :      1,
    "_node" :       "mzIVYCsqSWCG_M_ZffSs9Q",
```

然后返回值中的 `_explanation` 会包含在每一个入口，告诉你采用了哪种计算方式，并让你知道计算的结果以及其他详情：

```
"_explanation": { <1>
   "description": "weight(tweet:honeymoon in 0)
                   [PerFieldSimilarity], result of:",
   "value":       0.076713204,
   "details": [
      {
         "description": "fieldWeight in 0, product of:",
         "value":       0.076713204,
         "details": [
            {  <2>
               "description": "tf(freq=1.0), with freq of:",
               "value":       1,
               "details": [
                  {
                     "description": "termFreq=1.0",
                     "value":       1
                  }
               ]
            },
            { <3>
               "description": "idf(docFreq=1, maxDocs=1)",
               "value":       0.30685282
            },
            { <4>
               "description": "fieldNorm(doc=0)",
               "value":        0.25,
            }
         ]
      }
   ]
}
```

<1> `honeymoon` 相关性评分计算的总结

<2> 检索词频率

<3> 反向文档频率

<4> 字段长度准则

> 重要： 输出 `explain` 结果代价是十分昂贵的，它只能用作调试工具 --千万不要用于生产环境。

第一部分是关于计算的总结。告诉了我们 `"honeymoon"` 在 `tweet` 字段中的检索词频率/反向文档频率或 TF/IDF，（这里的文档 `0` 是一个内部的ID，跟我们没有关系，可以忽略。）

然后解释了计算的权重是如何计算出来的：

检索词频率::

```
检索词 `honeymoon` 在 `tweet` 字段中的出现次数。
```

反向文档频率::

```
检索词 `honeymoon` 在 `tweet` 字段在当前文档出现次数与索引中其他文档的出现总数的比率。
```

字段长度准则::

```
文档中 `tweet` 字段内容的长度 -- 内容越长, How long s the d field in this document -- the longer the field,
the smaller this number.
```

复杂的查询语句解释也非常复杂，但是包含的内容与上面例子大致相同。 通过这段描述我们可以了解搜索结果是如何产生的。

> 提示： JSON形式的explain描述是难以阅读的 但是转成 YAML 会好很多，只需要在参数中加上 `format=yaml`

Explain Api

文档是如何被匹配到的

当 `explain` 选项加到某一文档上时，它会告诉你为何这个文档会被匹配，以及一个文档为何没有被匹配。

请求路径为 `/index/type/id/_explain`，如下所示：

```
GET /us/tweet/12/_explain
{
   "query" : {
      "filtered" : {
         "filter" : { "term" :  { "user_id" : 2           }},
         "query" :  { "match" : { "tweet" :   "honeymoon" }}
      }
   }
}
```

除了上面我们看到的完整描述外，我们还可以看到这样的描述：

```
"failure to match filter: cache(user_id:[2 TO 2])"
```

也就是说我们的 `user_id` 过滤子句使该文档不能匹配到。

数据字段

本章的目的在于介绍关于ElasticSearch内部的一些运行情况。在这里我们先不介绍新的知识点， 数据字段是我们要经常查阅的内容之一，但我们使用的时候不必太在意。

当你对一个字段进行排序时，ElasticSearch 需要进入每个匹配到的文档得到相关的值。 倒排索引在用于搜索时是非常卓越的，但却不是理想的排序结构。

- 当搜索的时候，我们需要用检索词去遍历所有的文档。
- 当排序的时候，我们需要遍历文档中所有的值，我们需要做反倒序排列操作。

为了提高排序效率，ElasticSearch 会将所有字段的值加载到内存中，这就叫做"数据字段"。

> 重要： ElasticSearch将所有字段数据加载到内存中并不是匹配到的那部分数据。 而是索引下所有文档中的值，包括所有类型。

将所有字段数据加载到内存中是因为从硬盘反向倒排索引是非常缓慢的。尽管你这次请求需要的是某些文档中的部分数据，但你下个请求却需要另外的数据，所以将所有字段数据一次性加载到内存中是十分必要的。

ElasticSearch中的字段数据常被应用到以下场景：

- 对一个字段进行排序
- 对一个字段进行聚合
- 某些过滤，比如地理位置过滤
- 某些与字段相关的脚本计算

毫无疑问，这会消耗掉很多内存，尤其是大量的字符串数据 -- string字段可能包含很多不同的值，比如邮件内容。 值得庆幸的是，内存不足是可以通过横向扩展解决的，我们可以增加更多的节点到集群。

现在，你只需要知道字段数据是什么，和什么时候内存不足就可以了。 稍后我们会讲述字段数据到底消耗了多少内存，如何限制ElasticSearch可以使用的内存，以及如何预加载字段数据以提高用户体验。

分布式搜索的执行方式

在继续之前，我们将绕道讲一下搜索是如何在分布式环境中执行的。 它比我们之前讲的基础的增删改查(*create-read-update-delete* ， CRUD)请求要复杂一些。

> 注意：
>
> 本章的信息只是出于兴趣阅读，使用Elasticsearch并不需要理解和记住这里的所有细节。
>
> 阅读这一章只是增加对系统如何工作的了解，并让你知道这些信息以备以后参考，所以别淹没在细节里。

一个CRUD操作只处理一个单独的文档。文档的唯一性由 `_index` ， `_type` 和 `routing-value` （通常默认是该文档的 `_id` ）的组合来确定。这意味着我们可以准确知道集群中的哪个分片持有这个文档。

由于不知道哪个文档会匹配查询（文档可能存放在集群中的任意分片上），所以搜索需要一个更复杂的模型。一个搜索不得不通过查询每一个我们感兴趣的索引的分片副本，来看是否含有任何匹配的文档。

但是，找到所有匹配的文档只完成了这件事的一半。在搜索（ `search` ）API返回一页结果前，来自多个分片的结果必须被组合放到一个有序列表中。因此，搜索的执行过程分两个阶段，称为查询然后取回（*query then fetch*）。

查询阶段

在初始化查询阶段（*query phase*），查询被向索引中的每个分片副本（原本或副本）广播。每个分片在本地执行搜索并且建立了匹配document的优先队列（*priority queue*）。

> ## 优先队列
>
> 一个优先队列（*priority queue* is）只是一个存有前*n*个（*top-n*）匹配document的有序列表。这个优先队列的大小由分页参数from和size决定。例如，下面这个例子中的搜索请求要求优先队列要能够容纳100个document

```
GET /_search
{
    "from": 90,
    "size": 10
}
```

这个查询的过程被描述在图分布式搜索查询阶段中。

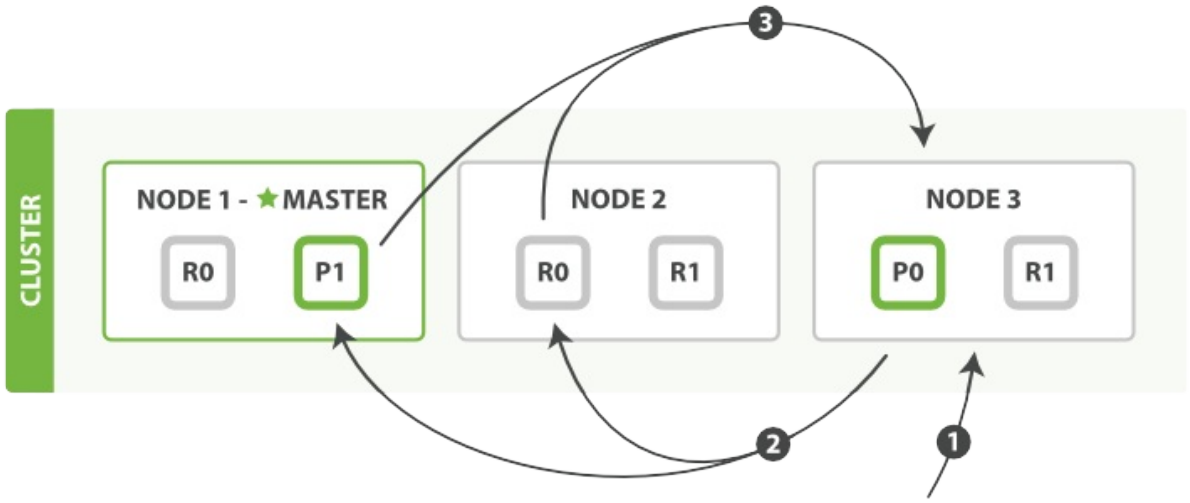


图1 分布式搜索查询阶段

查询阶段包含以下三步：

1.客户端发送一个 `search（搜索）` 请求给 `Node 3`，`Node 3` 创建了一个长度为 `from+size` 的空优先级队列。 2. `Node 3` 转发这个搜索请求到索引中每个分片的原本或副本。每个分片在本地执行这个查询并且结果将结果到一个大小为 `from+size` 的有序本地优先队列里去。 3.每个分片返回document的ID和它优先队列里的所有document的排序值给协调节点 `Node 3`。 `Node 3` 把这些值合并到自己的优先队列里产生全局排序结果。

当一个搜索请求被发送到一个节点Node，这个节点就变成了协调节点。这个节点的工作是向所有相关的分片广播搜索请求并且把它们的响应整合成一个全局的有序结果集。这个结果集会被返回给客户端。

第一步是向索引里的每个节点的分片副本广播请求。就像document的 `GET` 请求一样，搜索请求可以被每个分片的原本或任意副本处理。这就是更多的副本（当结合更多的硬件时）如何提高搜索的吞吐量的方法。对于后续请求，协调节点会轮询所有的分片副本以分摊负载。

每一个分片在本地执行查询和建立一个长度为 `from+size` 的有序优先队列——这个长度意味着它自己的结果数量就足够满足全局的请求要求。分片返回一个轻量级的结果列表给协调节点。只包含documentID值和排序需要用到的值，例如 `_score`。

协调节点将这些分片级的结果合并到自己的有序优先队列里。这个就代表了最终的全局有序结果集。到这里，查询阶段结束。

> ### 注意
>
> 一个索引可以由一个或多个原始分片组成，所以一个对于单个索引的搜索请求也需要能够把来自多个分片的结果组合起来。一个对于 多（*multiple*）或全部（*all*）索引的搜索的工作机制和这完全一致——仅仅是多了一些分片而已。

取回阶段

查询阶段辨别出那些满足搜索请求的document，但我们仍然需要取回那些document本身。这就是取回阶段的工作，如图分布式搜索的取回阶段所示。

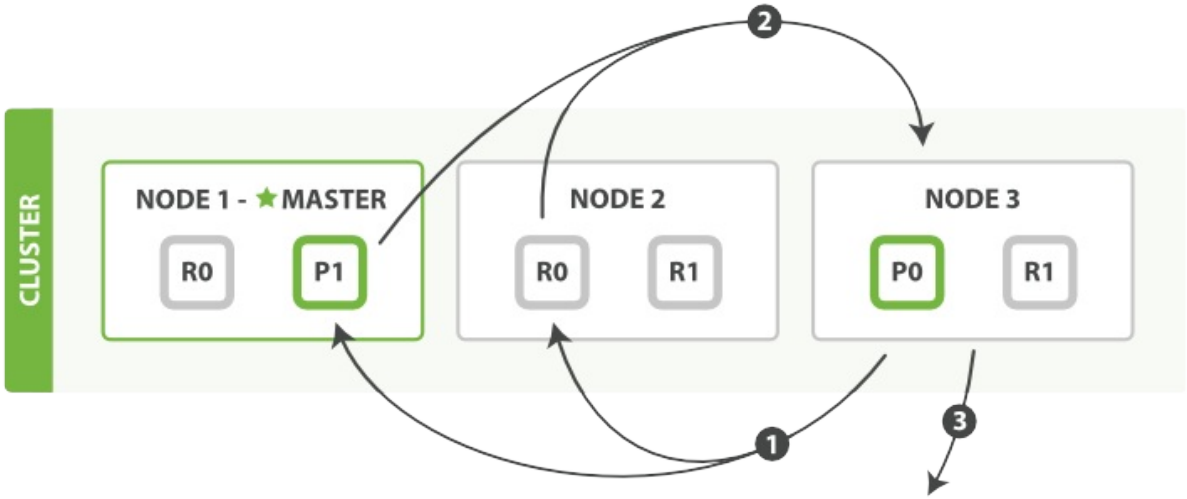


图2 分布式搜索取回阶段

分发阶段由以下步骤构成：

1.协调节点辨别出哪个document需要取回，并且向相关分片发出 `GET` 请求。

2.每个分片加载document并且根据需要丰富（*enrich*）它们，然后再将document返回协调节点。

3.一旦所有的document都被取回，协调节点会将结果返回给客户端。

协调节点先决定哪些document是实际（*actually*）需要取回的。例如，我们指定查询 `{ "from": 90, "size": 10 }`，那么前90条将会被丢弃，只有之后的10条会需要取回。这些document可能来自与原始查询请求相关的某个、某些或者全部分片。

协调节点为每个持有相关document的分片建立多点get请求然后发送请求到处理查询阶段的分片副本。

分片加载document主体—— `_source` field。如果需要，还会根据元数据丰富结果和高亮搜索片断。一旦协调节点收到所有结果，会将它们汇集到单一的回答响应里，这个响应将会返回给客户端。

深分页

查询然后取回过程虽然支持通过使用 `from` 和 `size` 参数进行分页，但是要在有限范围内（*within limited*）。还记得每个分片必须构造一个长度为 `from+size` 的优先队列吧，所有这些都要传回协调节点。这意味着协调节点要通过对 `分片数量 * (from + size)` 个document进行排序来找到正确的 `size` 个document。

根据document的数量，分片的数量以及所使用的硬件，对10,000到50,000条结果（1,000到5,000页）深分页是可行的。但是对于足够大的 `from` 值，排序过程将会变得非常繁重，会使用巨大量的CPU，内存和带宽。因此，强烈不建议使用深分页。

在实际中，“深分页者”也是很少的一部人。一般人会在翻了两三页后就停止翻页，并会更改搜索标准。那些不正常情况通常是机器人或者网络爬虫的行为。它们会持续不断地一页接着一页地获取页面直到服务器到底崩溃的边缘。

如果你确实需要从集群里获取大量documents，你可以通过设置搜索类型 `scan` 禁用排序，来高效地做这件事。这一点将在后面的章节讨论。

搜索选项

一些查询字符串（query-string）可选参数能够影响搜索过程。

preference（偏爱）

`preference` 参数允许你控制使用哪个分片或节点来处理搜索请求。它接受如下一些参数 `_primary`， `_primary_first`， `_local`， `_only_node:xyz`， `_prefer_node:xyz` 和 `_shards:2,3`。这些参数在文档搜索偏好（search preference）里有详细描述。

然而通常最有用的值是一些随机字符串，它们可以避免结果震荡问题（the *bouncing results* problem）。

结果震荡（**Bouncing Results**）

- 想像一下，你正在按照 `timestamp` 字段来对你的结果排序，并且有两个document有相同的timestamp。由于搜索请求是在所有有效的分片副本间轮询的，这两个document可能在原始分片里是一种顺序，在副本分片里是另一种顺序。
- 这就是被称为结果震荡（*bouncing results*）的问题：用户每次刷新页面，结果顺序会发生变化。避免这个问题方法是对于同一个用户总是使用同一个分片。方法就是使用一个随机字符串例如用户的会话ID（session ID）来设置 `preference` 参数。

timeout（超时）

通常，协调节点会等待接收所有分片的回答。如果有一个节点遇到问题，它会拖慢整个搜索请求。

`timeout` 参数告诉协调节点最多等待多久，就可以放弃等待而将已有结果返回。返回部分结果总比什么都没有好。

搜索请求的返回将会指出这个搜索是否超时，以及有多少分片成功答复了：

```
...
"timed_out":     true,  (1)
"_shards": {
   "total":      5,
   "successful": 4,
   "failed":     1     (2)
},
...
```

---

(1) 搜索请求超时。

(2) 五个分片中有一个没在超时时间内答复。

如果一个分片的所有副本都因为其他原因失败了——也许是因为硬件故障——这个也同样会反映在该答复的 `_shards` 部分里。

routing（路由选择）

在路由值那节里，我们解释了如何在建立索引时提供一个自定义的 `routing` 参数来保证所有相关的document（如属于单个用户的document）被存放在一个单独的分片中。在搜索时，你可以指定一个或多个 `routing` 值来限制只搜索那些分片而不是搜索index里的全部分片：

```
GET /_search?routing=user_1,user2
```

这个技术在设计非常大的搜索系统时就会派上用场了。我们在规模（scale）那一章里详细讨论它。

search_type（搜索类型）

虽然`query_then_fetch`是默认的搜索类型，但也可以根据特定目的指定其它的搜索类型，例如：

```
GET /_search?search_type=count
```

count（计数）

`count（计数）`搜索类型只有一个`query（查询）`的阶段。当不需要搜索结果只需要知道满足查询的document的数量时，可以使用这个查询类型。

query_and_fetch（查询并且取回）

`query_and_fetch（查询并且取回）`搜索类型将查询和取回阶段合并成一个步骤。这是一个内部优化选项，当搜索请求的目标只是一个分片时可以使用，例如指定了`routing（路由选择）`值时。虽然你可以手动选择使用这个搜索类型，但是这么做基本上不会有什么效果。

dfs_query_then_fetch 和 ***dfs_query_and_fetch***

`dfs`搜索类型有一个预查询的阶段，它会从全部相关的分片里取回项目频数来计算全局的项目频数。我们将在relevance-is-broken（相关性被破坏）里进一步讨论这个。

scan（扫描）

`scan（扫描）`搜索类型是和`scroll（滚屏）`API连在一起使用的，可以高效地取回巨大数量的结果。它是通过禁用排序来实现的。我们将在下一节*scan-and-scroll*（扫描和滚屏）里讨论它。

扫描和滚屏

`scan（扫描）`搜索类型是和`scroll（滚屏）`API一起使用来从Elasticsearch里高效地取回巨大数量的结果而不需要付出深分页的代价。

scroll（滚屏）

一个滚屏搜索允许我们做一个初始阶段搜索并且持续批量从Elasticsearch里拉取结果直到没有结果剩下。这有点像传统数据库里的*cursors*（游标）。

滚屏搜索会及时制作快照。这个快照不会包含任何在初始阶段搜索请求后对index做的修改。它通过将旧的数据文件保存在手边，所以可以保护index的样子看起来像搜索开始时的样子。

scan（扫描）

深度分页代价最高的部分是对结果的全局排序，但如果禁用排序，就能以很低的代价获得全部返回结果。为达成这个目的，可以采用`scan（扫描）`搜索模式。扫描模式让Elasticsearch不排序，只要分片里还有结果可以返回，就返回一批结果。

为了使用*scan-and-scroll*（扫描和滚屏），需要执行一个搜索请求，将`search_type`设置成`scan`，并且传递一个`scroll`参数来告诉Elasticsearch滚屏应该持续多长时间。

```
GET /old_index/_search?search_type=scan&scroll=1m (1)
{
    "query": { "match_all": {}},
    "size":  1000
}
```

（1）保持滚屏开启1分钟。

这个请求的应答没有包含任何命中的结果，但是包含了一个Base-64编码的`_scroll_id（滚屏id）`字符串。现在我们可以将`_scroll_id`传递给`_search/scroll`末端来获取第一批结果：

```
GET /_search/scroll?scroll=1m      (1)
c2Nhbjs1OzExODpRNV9aY1VyUVM4U0NMd2pjWlJ3YWlBOzExOTpRNV9aY1VyUVM4U0 <2>
NMd2pjWlJ3YWlBOzExNjpRNV9aY1VyUVM4U0NMd2pjWlJ3YWlBOzExNzpRNV9aY1Vy
UVM4U0NMd2pjWlJ3YWlBOzEyMDpRNV9aY1VyUVM4U0NMd2pjWlJ3YWlBOzE7dG90YW
xfaGl0czoxOw==
```

---

(1) 保持滚屏开启另一分钟。

(2) `_scroll_id`可以在body或者URL里传递，也可以被当做查询参数传递。

注意，要再次指定`?scroll=1m`。滚屏的终止时间会在我们每次执行滚屏请求时刷新，所以他只需要给我们足够的时间来处理当前批次的结果而不是所有的匹配查询的document。

这个滚屏请求的应答包含了第一批次的结果。虽然指定了一个1000的`size`，但是获得了更多的document。当扫描时，`size`被应用到每一个分片上，所以我们在每个批次里最多或获得`size * number_of_primary_shards（size*主分片数）`个document。

> 注意：
>
> 滚屏请求也会返回一个新的`_scroll_id`。每次做下一个滚屏请求时，必须传递前一次请求返回的`_scroll_id`。

如果没有更多的命中结果返回，就处理完了所有的命中匹配的document。

> 提示：
>
> 一些Elasticsearch官方客户端提供扫描和滚屏的小助手。小助手提供了一个对这个功能的简单封装。

索引管理

我们已经看到Elasticsearch如何在不需要任何预先计划和设置的情况下，轻松地开发一个新的应用。并且，在你想调整索引和搜索过程来更好地适应你特殊的使用需求前，不会花较长的时间。它包含几乎所有的和索引及类型相关的定制选项。在这一章，将介绍管理索引和类型映射的API以及最重要的设置。

创建索引

迄今为止，我们简单的通过添加一个文档的方式创建了一个索引。这个索引使用默认设置，新的属性通过动态映射添加到分类中。现在我们需要对这个过程有更多的控制：我们需要确保索引被创建在适当数量的分片上，在索引数据之前设置好分析器和类型映射。

为了达到目标，我们需要手动创建索引，在请求中加入所有设置和类型映射，如下所示：

```
PUT /my_index
{
    "settings": { ... any settings ... },
    "mappings": {
        "type_one": { ... any mappings ... },
        "type_two": { ... any mappings ... },
        ...
    }
```

事实上，你可以通过在 `config/elasticsearch.yml` 中添加下面的配置来防止自动创建索引。

```
action.auto_create_index: false
```

> **NOTE**
>
> 今后，我们将介绍怎样用【索引模板】来自动预先配置索引。这在索引日志数据时尤其有效： 你将日志数据索引在一个以日期结尾的索引上，第二天，一个新的配置好的索引会自动创建好。

删除索引

使用以下的请求来删除索引：

```
DELETE /my_index
```

你也可以用下面的方式删除多个索引

```
DELETE /index_one,index_two
DELETE /index_*
```

你甚至可以删除所有索引

```
DELETE /_all
```

索引设置

你可以通过很多种方式来自定义索引行为，你可以阅读Index Modules reference documentation，但是：

提示: Elasticsearch 提供了优化好的默认配置。除非你明白这些配置的行为和为什么要这么做，请不要修改这些配置。

下面是两个最重要的设置：

`number_of_shards`

> 定义一个索引的主分片个数，默认值是 `5`。这个配置在索引创建后不能修改。

`number_of_replicas`

> 每个主分片的复制分片个数，默认是 `1`。这个配置可以随时在活跃的索引上修改。

例如，我们可以创建只有一个主分片，没有复制分片的小索引。

```
PUT /my_temp_index
{
    "settings": {
        "number_of_shards" :   1,
        "number_of_replicas" : 0
    }
}
```

然后，我们可以用 `update-index-settings` API 动态修改复制分片个数：

```
PUT /my_temp_index/_settings
{
    "number_of_replicas": 1
}
```

配置分析器

第三个重要的索引设置是 `analysis` 部分，用来配置已存在的分析器或创建自定义分析器来定制化你的索引。

在【分析器介绍】中，我们介绍了一些内置的分析器，用于将全文字符串转换为适合搜索的倒排索引。

`standard` 分析器是用于全文字段的默认分析器，对于大部分西方语系来说是一个不错的选择。它考虑了以下几点：

- `standard` 分词器，在词层级上分割输入的文本。
- `standard` 表征过滤器，被设计用来整理分词器触发的所有表征（但是目前什么都没做）。
- `lowercase` 表征过滤器，将所有表征转换为小写。
- `stop` 表征过滤器，删除所有可能会造成搜索歧义的停用词，如 `a`， `the`， `and`， `is`。

默认情况下，停用词过滤器是被禁用的。如需启用它，你可以通过创建一个基于 `standard` 分析器的自定义分析器，并且设置 `stopwords` 参数。可以提供一个停用词列表，或者使用一个特定语言的预定停用词列表。

在下面的例子中，我们创建了一个新的分析器，叫做 `es_std`，并使用预定义的西班牙语停用词：

```
PUT /spanish_docs
{
    "settings": {
        "analysis": {
            "analyzer": {
                "es_std": {
                    "type":      "standard",
                    "stopwords": "_spanish_"
                }
            }
        }
    }
}
```

`es_std` 分析器不是全局的，它仅仅存在于我们定义的 `spanish_docs` 索引中。为了用 `analyze` API 来测试它，我们需要使用特定的索引名。

```
GET /spanish_docs/_analyze?analyzer=es_std
El veloz zorro marrón
```

下面简化的结果中显示停用词 `El` 被正确的删除了：

```
{
  "tokens" : [
    { "token" :    "veloz",   "position" : 2 },
    { "token" :    "zorro",   "position" : 3 },
    { "token" :    "marrón",  "position" : 4 }
  ]
}
```

自定义分析器

虽然 Elasticsearch 内置了一系列的分析器，但是真正的强大之处在于定制你自己的分析器。你可以通过在配置文件中组合字符过滤器，分词器和表征过滤器，来满足特定数据的需求。

在 【分析器介绍】 中，我们提到 分析器 是三个顺序执行的组件的结合（字符过滤器，分词器，表征过滤器）。

字符过滤器

> 字符过滤器是让字符串在被分词前变得更加“整洁”。例如，如果我们的文本是 HTML 格式，它可能会包含一些我们不想被索引的 HTML 标签，诸如 `<p>` 或 `<div>`。
>
> 我们可以使用 `html_strip` 字符过滤器 来删除所有的 HTML 标签，并且将 HTML 实体转换成对应的 Unicode 字符，比如将 `&Aacute;` 转成 `Á`。
>
> 一个分析器可能包含零到多个字符过滤器。

分词器

> 一个分析器 必须 包含一个分词器。分词器将字符串分割成单独的词（terms）或表征（tokens）。`standard` 分析器使用 `standard` 分词器将字符串分割成单独的字词，删除大部分标点符号，但是现存的其他分词器会有不同的行为特征。
>
> 例如，`keyword` 分词器输出和它接收到的相同的字符串，不做任何分词处理。[`whitespace` 分词器]只通过空格来分割文本。[`pattern` 分词器]可以通过正则表达式来分割文本。

表征过滤器

> 分词结果的 表征流 会根据各自的情况，传递给特定的表征过滤器。
>
> 表征过滤器可能修改，添加或删除表征。我们已经提过 `lowercase` 和 `stop` 表征过滤器，但是 Elasticsearch 中有更多的选择。`stemmer` 表征过滤器将单词转化为他们的根形态（root form）。`ascii_folding` 表征过滤器会删除变音符号，比如从 `très` 转为 `tres`。 `ngram` 和 `edge_ngram` 可以让表征更适合特殊匹配情况或自动完成。

在【深入搜索】中，我们将举例介绍如何使用这些分词器和过滤器。但是首先，我们需要阐述一下如何创建一个自定义分析器

创建自定义分析器

与索引设置一样，我们预先配置好 `es_std` 分析器，我们可以再 `analysis` 字段下配置字符过滤器，分词器和表征过滤器：

```
PUT /my_index
{
    "settings": {
        "analysis": {
            "char_filter": { ... custom character filters ... },
            "tokenizer":   { ...    custom tokenizers     ... },
            "filter":      { ...   custom token filters   ... },
            "analyzer":    { ...    custom analyzers      ... }
        }
    }
}
```

作为例子，我们来配置一个这样的分析器：

1. 用 `html_strip` 字符过滤器去除所有的 HTML 标签
2. 将 `&` 替换成 `and`，使用一个自定义的 `mapping` 字符过滤器

```
"char_filter": {
    "&_to_and": {
        "type":       "mapping",
        "mappings": [ "&=> and "]
    }
}
```

1. 使用 `standard` 分词器分割单词
2. 使用 `lowercase` 表征过滤器将词转为小写
3. 用 `stop` 表征过滤器去除一些自定义停用词。

```
"filter": {
    "my_stopwords": {
        "type":        "stop",
        "stopwords": [ "the", "a" ]
    }
}
```

根据以上描述来将预定义好的分词器和过滤器组合成我们的分析器：

```
"analyzer": {
    "my_analyzer": {
        "type":           "custom",
        "char_filter":  [ "html_strip", "&_to_and" ],
        "tokenizer":      "standard",
        "filter":       [ "lowercase", "my_stopwords" ]
    }
}
```

用下面的方式可以将以上请求合并成一条：

```
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" ]
            }}
}}}
```

创建索引后，用 `analyze` API 来测试新的分析器：

```
GET /my_index/_analyze?analyzer=my_analyzer
The quick & brown fox
```

下面的结果证明我们的分析器能正常工作了：

```
{
```

```
  "tokens" : [
      { "token" :   "quick",    "position" : 2 },
      { "token" :   "and",      "position" : 3 },
      { "token" :   "brown",    "position" : 4 },
      { "token" :   "fox",      "position" : 5 }
    ]
}
```

除非我们告诉 Elasticsearch 在哪里使用，否则分析器不会起作用。我们可以通过下面的映射将它应用在一个 `string` 类型的字段上：

```
PUT /my_index/_mapping/my_type
{
    "properties": {
        "title": {
            "type":      "string",
            "analyzer":  "my_analyzer"
        }
    }
}
```

类型和映射

类型 在 Elasticsearch 中表示一组相似的文档。类型 由一个 名称（比如 `user` 或 `blogpost`）和一个类似数据库表结构的映射组成，描述了文档中可能包含的每个字段的 属性，数据类型（比如 `string`，`integer` 或 `date`），和是否这些字段需要被 Lucene 索引或储存。

在【文档】一章中，我们说过类型类似关系型数据库中的表格，一开始你可以这样做类比，但是现在值得更深入阐释一下什么是类型，且在 Lucene 中是怎么实现的。

Lucene 如何处理文档

Lucene 中，一个文档由一组简单的键值对组成，一个字段至少需要有一个值，但是任何字段都可以有多个值。类似的，一个单独的字符串可能在分析过程中被转换成多个值。Lucene 不关心这些值是字符串，数字或日期，所有的值都被当成 不透明字节

当我们在 Lucene 中索引一个文档时，每个字段的值都被加到相关字段的倒排索引中。你也可以选择将原始数据 储存 起来以备今后取回。

类型是怎么实现的

Elasticsearch 类型是在这个简单基础上实现的。一个索引可能包含多个类型，每个类型有各自的映射和文档，保存在同一个索引中。

因为 Lucene 没有文档类型的概念，每个文档的类型名被储存在一个叫 `_type` 的元数据字段上。当我们搜索一种特殊类型的文档时，Elasticsearch 简单的通过 `_type` 字段来过滤出这些文档。

Lucene 同样没有映射的概念。映射是 Elasticsearch 将复杂 JSON 文档映射成 Lucene 需要的扁平化数据的方式。

例如，`user` 类型中 `name` 字段的映射声明这个字段是一个 `string` 类型，在被加入倒排索引之前，它的数据需要通过 `whitespace` 分析器来分析。

```
"name": {
    "type":     "string",
    "analyzer": "whitespace"
}
```

预防类型陷阱

事实上不同类型的文档可以被加到同一个索引里带来了一些预想不到的困难。

想象一下我们的索引中有两种类型：`blog_en` 表示英语版的博客，`blog_es` 表示西班牙语版的博客。两种类型都有 `title` 字段，但是其中一种类型使用 `english` 分析器，另一种使用 `spanish` 分析器。

使用下面的查询就会遇到问题：

```
GET /_search
{
    "query": {
        "match": {
            "title": "The quick brown fox"
        }
    }
}
```

我们在两种类型中搜索 `title` 字段，首先需要分析查询语句，但是应该使用哪种分析器呢，`spanish` 还是 `english`？Elasticsearch 会采用第一个被找到的 `title` 字段使用的分析器，这对于这个字段的文档来说是正确的，但对另一个来说却是错误的。

我们可以通过给字段取不同的名字来避免这种错误 —— 比如，用 `title_en` 和 `title_es` 。或者在查询中明确包含各自的类型名。

```
GET /_search
{
    "query": {
        "multi_match": { <1>
            "query":    "The quick brown fox",
            "fields": [ "blog_en.title", "blog_es.title" ]
        }
    }
}
```

<1> `multi_match` 查询在多个字段上执行 `match` 查询并一起返回结果。

新的查询中 `english` 分析器用于 `blog_en.title` 字段， `spanish` 分析器用于 `blog_es.title` 字段，然后通过综合得分组合两种字段的结果。

这种办法对具有相同数据类型的字段有帮助，但是想象一下如果你将下面两个文档加入同一个索引，会发生什么：

- 类型: user

```
{ "login": "john_smith" }
```

- 类型: event

```
{ "login": "2014-06-01" }
```

Lucene 不在乎一个字段是字符串而另一个字段是日期，它会一视同仁的索引这两个字段。

然而，假如我们试图 排序 `event.login` 字段，Elasticsearch 需要将 `login` 字段的值加载到内存中。像我们在 【字段数据介绍】中提到的，它将 任意文档 的值加入索引而不管它们的类型。

它会尝试加载这些值为字符串或日期，取决于它遇到的第一个 `login` 字段。这可能会导致预想不到的结果或者以失败告终。

提示：为了保证你不会遇到这些冲突，建议在同一个索引的每一个类型中，确保用同样的方式映射同名的字段

根对象

映射的最高一层被称为 根对象，它可能包含下面几项：

- 一个 *properties* 节点，列出了文档中可能包含的每个字段的映射
- 多个元数据字段，每一个都以下划线开头，例如 `_type`，`_id` 和 `_source`
- 设置项，控制如何动态处理新的字段，例如 `analyzer`，`dynamic_date_formats` 和 `dynamic_templates`。
- 其他设置，可以同时应用在根对象和其他 `object` 类型的字段上，例如 `enabled`，`dynamic` 和 `include_in_all`

属性

我们已经在【核心字段】和【复合核心字段】章节中介绍过文档字段和属性的三个最重要的设置：

`type`：字段的数据类型，例如 `string` 和 `date`

`index`：字段是否应当被当成全文来搜索（`analyzed`），或被当成一个准确的值（`not_analyzed`），还是完全不可被搜索（`no`）

`analyzer`：确定在索引和或搜索时全文字段使用的 `分析器`。

我们将在下面的章节中介绍其他字段，例如 `ip`，`geo_point` 和 `geo_shape`

元数据：_source 字段

默认情况下，Elasticsearch 用 JSON 字符串来表示文档主体保存在 `_source` 字段中。像其他保存的字段一样，`_source` 字段也会在写入硬盘前压缩。

这几乎始终是需要的功能，因为：

- 搜索结果中能得到完整的文档 —— 不需要额外去别的数据源中查询文档
- 如果缺少 `_source` 字段，部分 `更新` 请求不会起作用
- 当你的映射有变化，而且你需要重新索引数据时，你可以直接在 Elasticsearch 中操作而不需要重新从别的数据源中取回数据。
- 你可以从 `_source` 中通过 `get` 或 `search` 请求取回部分字段，而不是整个文档。
- 这样更容易排查错误，因为你可以准确的看到每个文档中包含的内容，而不是只能从一堆 ID 中猜测他们的内容。

即便如此，存储 `_source` 字段还是要占用硬盘空间的。假如上面的理由对你来说不重要，你可以用下面的映射禁用 `_source` 字段：

```
PUT /my_index
{
    "mappings": {
        "my_type": {
            "_source": {
                "enabled":  false
            }
        }
    }
}
```

在搜索请求中你可以通过限定 `_source` 字段来请求指定字段：

```
GET /_search
{
    "query":   { "match_all": {}},
    "_source": [ "title", "created" ]
}
```

这些字段会从 `_source` 中提取出来，而不是返回整个 `_source` 字段。

> 储存字段
>
> 除了索引字段的值，你也可以选择 `储存` 字段的原始值以备日后取回。使用 Lucene 做后端的用户用储存字段来选择搜索结果的返回值，事实上，`_source` 字段就是一个储存字段。
>
> 在 Elasticsearch 中，单独设置储存字段不是一个好做法。完整的文档已经被保存在 `_source` 字段中。通常最好的办法会是使用 `_source` 参数来过滤你需要的字段。

元数据：_all 字段

在【简单搜索】中，我们介绍了 `_all` 字段：一个所有其他字段值的特殊字符串字段。`query_string` 在没有指定字段时默认用 `_all` 字段查询。

`_all` 字段在新应用的探索阶段比较管用，当你还不清楚最终文档的结构时，可以将任何查询用于这个字段，就有机会得到你想要的文档：

```
GET /_search
{
    "match": {
        "_all": "john smith marketing"
    }
}
```

随着你应用的发展，搜索需求会变得更加精准。你会越来越少的使用 `_all` 字段。`_all` 是一种简单粗暴的搜索方式。通过查询独立的字段，你能更灵活，强大和精准的控制搜索结果，提高相关性。

提示

【相关性算法】考虑的一个最重要的原则是字段的长度：字段越短，就越重要。在较短的 `title` 字段中的短语会比较长的 `content` 字段中的短语显得更重要。而字段间的这种差异在 `_all` 字段中就不会出现

如果你决定不再使用 `_all` 字段，你可以通过下面的映射禁用它：

```
PUT /my_index/_mapping/my_type
{
    "my_type": {
        "_all": { "enabled": false }
    }
}
```

通过 `include_in_all` 选项可以控制字段是否要被包含在 `_all` 字段中，默认值是 `true`。在一个对象上设置 `include_in_all` 可以修改这个对象所有字段的默认行为。

你可能想要保留 `_all` 字段来查询所有特定的全文字段，例如 `title`，`overview`，`summary` 和 `tags`。相对于完全禁用 `_all` 字段，你可以先默认禁用 `include_in_all` 选项，而选定字段上启用 `include_in_all`。

```
PUT /my_index/my_type/_mapping
{
    "my_type": {
        "include_in_all": false,
        "properties": {
            "title": {
                "type":           "string",
                "include_in_all": true
            },
            ...
        }
    }
}
```

谨记 `_all` 字段仅仅是一个经过分析的 `string` 字段。它使用默认的分析器来分析它的值，而不管这值本来所在的字段指定的分析器。而且像所有 `string` 类型字段一样，你可以配置 `_all` 字段使用的分析器：

```
PUT /my_index/my_type/_mapping
{
    "my_type": {
        "_all": { "analyzer": "whitespace" }
    }
}
```

文档 ID

文档唯一标识由四个元数据字段组成：

`_id` ：文档的字符串 ID

`_type` ：文档的类型名

`_index` ：文档所在的索引

`_uid` ： `_type` 和 `_id` 连接成的 `type#id`

默认情况下， `_uid` 是被保存（可取回）和索引（可搜索）的。 `_type` 字段被索引但是没有保存， `_id` 和 `_index` 字段则既没有索引也没有储存，它们并不是真实存在的。

尽管如此，你仍然可以像真实字段一样查询 `_id` 字段。Elasticsearch 使用 `_uid` 字段来追溯 `_id` 。虽然你可以修改这些字段的 `index` 和 `store` 设置，但是基本上不需要这么做。

`_id` 字段有一个你可能用得到的设置： `path` 设置告诉 Elasticsearch 它需要从文档本身的哪个字段中生成 `_id`

```
PUT /my_index
{
    "mappings": {
        "my_type": {
            "_id": {
                "path": "doc_id" <1>
            },
            "properties": {
                "doc_id": {
                    "type":   "string",
                    "index":  "not_analyzed"
                }
            }
        }
    }
}
```

<1> 从 `doc_id` 字段生成 `_id`

然后，当你索引一个文档时：

```
POST /my_index/my_type
{
    "doc_id": "123"
}
```

`_id` 值由文档主体的 `doc_id` 字段生成。

```
{
    "_index":   "my_index",
    "_type":    "my_type",
    "_id":      "123", <1>
    "_version": 1,
    "created":  true
}
```

<1> `_id` 正确的生成了。

警告：虽然这样很方便，但是注意它对 `bulk` 请求（见【bulk 格式】）有个轻微的性能影响。处理请求的节点将不能仅靠解析元数据行来决定将请求分配给哪一个分片，而需要解析整个文档主体。

动态映射

当 Elasticsearch 遭遇一个位置的字段时，它通过【动态映射】来确定字段的数据类型且自动将该字段加到类型映射中。

有时这是理想的行为，有时却不是。或许你不知道今后会有哪些字段加到文档中，但是你希望它们能自动被索引。或许你仅仅想忽略它们。特别是当你使用 Elasticsearch 作为主数据源时，你希望未知字段能抛出一个异常来警示你。

幸运的是，你可以通过 `dynamic` 设置来控制这些行为，它接受下面几个选项：

`true` ：自动添加字段（默认）

`false` ：忽略字段

`strict` ：当遇到未知字段时抛出异常

`dynamic` 设置可以用在根对象或任何 `object` 对象上。你可以将 `dynamic` 默认设置为 `strict` ，而在特定内部对象上启用它：

```
PUT /my_index
{
    "mappings": {
        "my_type": {
            "dynamic":      "strict", <1>
            "properties": {
                "title":  { "type": "string"},
                "stash":  {
                    "type":     "object",
                    "dynamic":  true <2>
                }
            }
        }
    }
}
```

<1> 当遇到未知字段时， `my_type` 对象将会抛出异常

<2> `stash` 对象会自动创建字段

通过这个映射，你可以添加一个新的可搜索字段到 `stash` 对象中：

```
PUT /my_index/my_type/1
{
    "title":   "This doc adds a new field",
    "stash": { "new_field": "Success!" }
}
```

但是在顶层做同样的操作则会失败：

```
PUT /my_index/my_type/1
{
    "title":     "This throws a StrictDynamicMappingException",
    "new_field": "Fail!"
}
```

备注：将 `dynamic` 设置成 `false` 完全不会修改 `_source` 字段的内容。 `_source` 将仍旧保持你索引时的完整 JSON 文档。然而，没有被添加到映射的未知字段将不可被搜索。

自定义动态索引

如果你想在运行时的增加新的字段，你可能会开启动态索引。虽然有时动态映射的 `规则` 显得不那么智能，幸运的是我们可以通过设置来自定义这些规则。

日期检测

当 Elasticsearch 遇到一个新的字符串字段时，它会检测这个字段是否包含一个可识别的日期，比如 `2014-01-01`。如果它看起来像一个日期，这个字段会被作为 `date` 类型添加，否则，它会被作为 `string` 类型添加。

有些时候这个规则可能导致一些问题。想象你有一个文档长这样：

```
{ "note": "2014-01-01" }
```

假设这是第一次见到 `note` 字段，它会被添加为 `date` 字段，但是如果下一个文档像这样：

```
{ "note": "Logged out" }
```

这显然不是一个日期，但为时已晚。这个字段已经被添加为日期类型，这个 `不合法的日期` 将引发异常。

日期检测可以通过在根对象上设置 `date_detection` 为 `false` 来关闭：

```
PUT /my_index
{
    "mappings": {
        "my_type": {
            "date_detection": false
        }
    }
}
```

使用这个映射，字符串将始终是 `string` 类型。假如你需要一个 `date` 字段，你得手动添加它。

提示：

Elasticsearch 判断字符串为日期的规则可以通过 `dynamic_date_formats` 配置 来修改。

动态模板

使用 `dynamic_templates`，你可以完全控制新字段的映射，你设置可以通过字段名或数据类型应用一个完全不同的映射。

每个模板都有一个名字用于描述这个模板的用途，一个 `mapping` 字段用于指明这个映射怎么使用，和至少一个参数（例如 `match`）来定义这个模板适用于哪个字段。

模板按照顺序来检测，第一个匹配的模板会被启用。例如，我们给 `string` 类型字段定义两个模板：

- `es`：字段名以 `_es` 结尾需要使用 `spanish` 分析器。
- `en`：所有其他字段使用 `english` 分析器。

我们将 `es` 模板放在第一位，因为它比匹配所有字符串的 `en` 模板更特殊一点

```
PUT /my_index
{
    "mappings": {
        "my_type": {
            "dynamic_templates": [
```

```
                { "es": {
                      "match":              "*_es", <1>
                      "match_mapping_type": "string",
                      "mapping": {
                          "type":           "string",
                          "analyzer":       "spanish"
                      }
                }},
                { "en": {
                      "match":              "*", <2>
                      "match_mapping_type": "string",
                      "mapping": {
                          "type":           "string",
                          "analyzer":       "english"
                      }
                }}
            ]
}}}
```

<1> 匹配字段名以 `_es` 结尾的字段.

<2> 匹配所有字符串类型字段。

`match_mapping_type` 允许你限制模板只能使用在特定的类型上，就像由标准动态映射规则检测的一样，（例如 `strong` 和 `long`）

`match` 参数只匹配字段名， `path_match` 参数则匹配字段在一个对象中的完整路径，所以 `address.*.name` 规则将匹配一个这样的字段：

```
{
    "address": {
        "city": {
            "name": "New York"
        }
    }
}
```

`unmatch` 和 `path_unmatch` 规则将用于排除未被匹配的字段。

更多选项见根对象参考文档

默认映射

通常，一个索引中的所有类型具有共享的字段和设置。用 `_default_` 映射来指定公用设置会更加方便，而不是每次创建新的类型时重复操作。 `_default` 映射像新类型的模板。所有在 `_default_` 映射 之后 的类型将包含所有的默认设置，除非在自己的类型映射中明确覆盖这些配置。

例如，我们可以使用 `_default_` 映射对所有类型禁用 `_all` 字段，而只在 `blog` 字段上开启它：

```
PUT /my_index
{
    "mappings": {
        "_default_": {
            "_all": { "enabled":  false }
        },
        "blog": {
            "_all": { "enabled":  true  }
        }
    }
}
```

`_default_` 映射也是定义索引级别的动态模板的好地方。

重新索引数据

虽然你可以给索引添加新的类型，或给类型添加新的字段，但是你不能添加新的分析器或修改已有字段。假如你这样做，已被索引的数据会变得不正确而你的搜索也不会正常工作。

修改在已存在的数据最简单的方法是重新索引：创建一个新配置好的索引，然后将所有的文档从旧的索引复制到新的上。

`_source` 字段的一个最大的好处是你已经在 Elasticsearch 中有了完整的文档，你不再需要从数据库中重建你的索引，这样通常会比较慢。

为了更高效的索引旧索引中的文档，使用【scan-scoll】来批量读取旧索引的文档，然后将通过【bulk API】来将它们推送给新的索引。

批量重新索引：

你可以在同一时间执行多个重新索引的任务，但是你显然不愿意它们的结果有重叠。所以，可以将重建大索引的任务通过日期或时间戳字段拆分成较小的任务：

```
GET /old_index/_search?search_type=scan&scroll=1m
{
    "query": {
        "range": {
            "date": {
                "gte":  "2014-01-01",
                "lt":   "2014-02-01"
            }
        }
    },
    "size":  1000
}
```

假如你继续在旧索引上做修改，你可能想确保新增的文档被加到了新的索引中。这可以通过重新运行重建索引程序来完成，但是记得只要过滤出上次执行后新增的文档就行了。

索引别名和零停机时间

前面提到的重新索引过程中的问题是必须更新你的应用，来使用另一个索引名。索引别名正是用来解决这个问题的！

索引 别名 就像一个快捷方式或软连接，可以指向一个或多个索引，也可以给任何需要索引名的 API 使用。别名带给我们极大的灵活性，允许我们做到：

- 在一个运行的集群上无缝的从一个索引切换到另一个
- 给多个索引分类（例如，`last_three_months`）
- 给索引的一个子集创建 `视图`

我们以后会讨论更多别名的使用场景。现在我们将介绍用它们怎么在零停机时间内从旧的索引切换到新的索引。

这里有两种管理别名的途径：`_alias` 用于单个操作，`_aliases` 用于原子化多个操作。

在这一章中，我们假设你的应用采用一个叫 `my_index` 的索引。而事实上，`my_index` 是一个指向当前真实索引的别名。真实的索引名将包含一个版本号：`my_index_v1`，`my_index_v2` 等等。

开始，我们创建一个索引 `my_index_v1`，然后将别名 `my_index` 指向它：

```
PUT /my_index_v1 <1>
PUT /my_index_v1/_alias/my_index <2>
```

<1> 创建索引 `my_index_v1`。

<2> 将别名 `my_index` 指向 `my_index_v1`。

你可以检测这个别名指向哪个索引：

```
GET /*/_alias/my_index
```

或哪些别名指向这个索引：

```
GET /my_index_v1/_alias/*
```

两者都将返回下列值：

```
{
    "my_index_v1" : {
        "aliases" : {
            "my_index" : { }
        }
    }
}
```

然后，我们决定修改索引中一个字段的映射。当然我们不能修改现存的映射，索引我们需要重新索引数据。首先，我们创建有新的映射的索引 `my_index_v2`。

```
PUT /my_index_v2
{
    "mappings": {
        "my_type": {
            "properties": {
                "tags": {
                    "type":   "string",
                    "index":  "not_analyzed"
```

```
                }
            }
        }
    }
}
```

然后我们从将数据从 `my_index_v1` 迁移到 `my_index_v2`，下面的过程在【重新索引】中描述过了。一旦我们认为数据已经被正确的索引了，我们就将别名指向新的索引。

别名可以指向多个索引，所以我们需要在新索引中添加别名的同时从旧索引中删除它。这个操作需要原子化，所以我们需要用 `_aliases` 操作：

```
POST /_aliases
{
    "actions": [
        { "remove": { "index": "my_index_v1", "alias": "my_index" }},
        { "add":    { "index": "my_index_v2", "alias": "my_index" }}
    ]
}
```

这样，你的应用就从旧索引迁移到了新的，而没有停机时间。

提示：

即使你认为现在的索引设计已经是完美的了，当你的应用在生产环境使用时，还是有可能在今后有一些改变的。

所以请做好准备：在应用中使用别名而不是索引。然后你就可以在任何时候重建索引。别名的开销很小，应当广泛使用。

入门

在分布式集群中，我们介绍了分片，把它描述为底层的工作单元。但分片到底是什么，它怎样工作？在这章节，我们将回答这些问题：

- 为什么搜索是近实时的？
- 为什么文档的CRUD操作是实时的？
- ES怎样保证更新持久化，即使断电也不会丢失？
- 为什么删除文档不会立即释放空间？
- 什么是`refresh, flush, optimize API`，以及什么时候你该使用它们？

为了理解分片如何工作，最简单的方式是从一堂历史课开始。我们将会看下，为了提供一个有近实时搜索和分析功能的分布式、持久化的搜索引擎需要解决哪些问题。

> 内容提示：
>
> 这章的内容是为了满足你的兴趣。为了使用ES，你不需要懂得并记住所有细节。阅读这章是为了感受下ES内部是如何运转的以及相关信息在哪，以备不时之需。但是不要被这些细节吓到。

使文本可以被搜索

第一个不得不解决的挑战是如何让文本变得可搜索。在传统的数据库中，一个字段存一个值，但是这对于全文搜索是不足的。想要让文本中的每个单词都可以被搜索，这意味这数据库需要存多个值。

支持一个字段多个值的最佳数据结构是倒排索引。倒排索引包含了出现在所有文档中唯一的值或词的有序列表，以及每个词所属的文档列表。

```
Term  | Doc 1 | Doc 2 | Doc 3 | ...
-----------------------------------
brown |   X   |       |  X    | ...
fox   |   X   |   X   |  X    | ...
quick |   X   |   X   |       | ...
the   |   X   |       |  X    | ...
```

> 注意
>
> 当讨论倒排索引时，我们说的是把文档加入索引。因为之前，一个倒排索引是用来索引整个非结构化的文本文档。ES的中文档是一个结构化的JSON文档。实际上，每一个JSON文档中被索引的字段都有它自己的倒排索引。

倒排索引存储了比包含了一个特定term的文档列表多地多的信息。它可能存储包含每个term的文档数量，一个term出现在指定文档中的频次，每个文档中term的顺序，每个文档的长度，所有文档的平均长度，等等。这些统计信息让Elasticsearch知道哪些term更重要，哪些文档更重要，也就是相关性。

需要意识到，为了实现倒排索引预期的功能，它必须要知道集合中所有的文档。

在全文检索的早些时候，会为整个文档集合建立一个大索引，并且写入磁盘。只有新的索引准备好了，它就会替代旧的索引，最近的修改才可以被检索。

不可变性

写入磁盘的倒排索引是不可变的，它有如下好处：

- 不需要锁。如果从来不需要更新一个索引，就不必担心多个程序同时尝试修改。
- 一旦索引被读入文件系统的缓存(译者:在内存)，它就一直在那儿，因为不会改变。只要文件系统缓存有足够的空间，大部分的读会直接访问内存而不是磁盘。这有助于性能提升。
- 在索引的声明周期内，所有的其他缓存都可用。它们不需要在每次数据变化了都重建，因为数据不会变。
- 写入单个大的倒排索引，可以压缩数据，较少磁盘IO和需要缓存索引的内存大小。

当然，不可变的索引有它的缺点，首先是它不可变！你不能改变它。如果想要搜索一个新文档，必须重见整个索引。这不仅严重限制了一个索引所能装下的数据，还有一个索引可以被更新的频次。

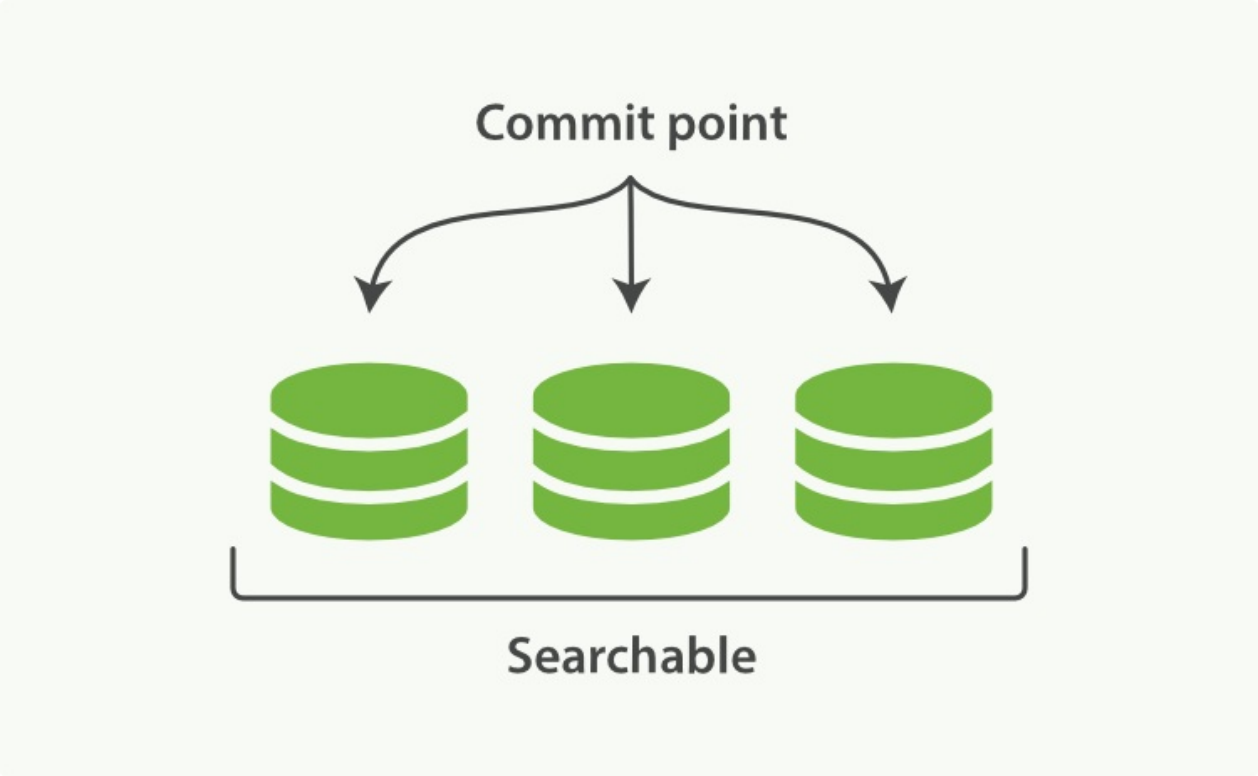
动态索引

下一个需要解决的问题是如何在保持不可变好处的同时更新倒排索引。答案是，使用多个索引。

不是重写整个倒排索引，而是增加额外的索引反映最近的变化。每个倒排索引都可以按顺序查询，从最老的开始，最后把结果聚合。

Elasticsearch底层依赖的Lucene，引入了 `per-segment search` 的概念。一个段(segment)是有完整功能的倒排索引，但是现在Lucene中的索引指的是段的集合，再加上提交点(commit point，包括所有段的文件)，如图**1**所示。新的文档，在被写入磁盘的段之前，首先写入内存区的索引缓存，如图**2**、图**3**所示。

图**1**：一个提交点和三个索引的**Lucene**



> 索引vs分片
>
> 为了避免混淆，需要说明，Lucene索引是Elasticsearch中的分片，Elasticsearch中的索引是分片的集合。当Elasticsearch搜索索引时，它发送查询请求给该索引下的所有分片，然后过滤这些结果，聚合成全局的结果。

一个 `per-segment search` 如下工作:

1. 新的文档首先写入内存区的索引缓存。
2. 不时，这些buffer被提交：
   - 一个新的段——额外的倒排索引——写入磁盘。
   - 新的提交点写入磁盘，包括新段的名称。
   - 磁盘是fsync'ed(文件同步)——所有写操作等待文件系统缓存同步到磁盘，确保它们可以被物理写入。
3. 新段被打开，它包含的文档可以被检索
4. 内存的缓存被清除，等待接受新的文档。

图**2**：内存缓存区有即将提交文档的**Lucene**索引

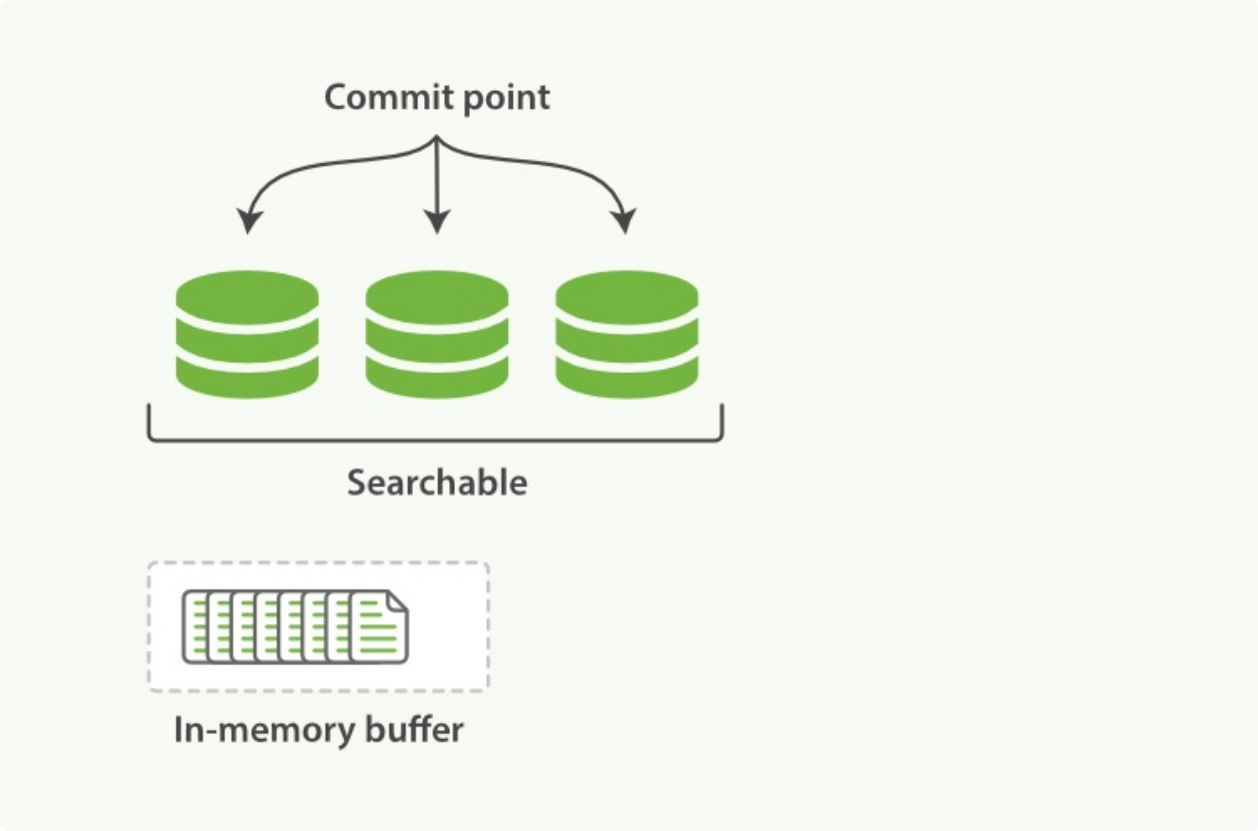
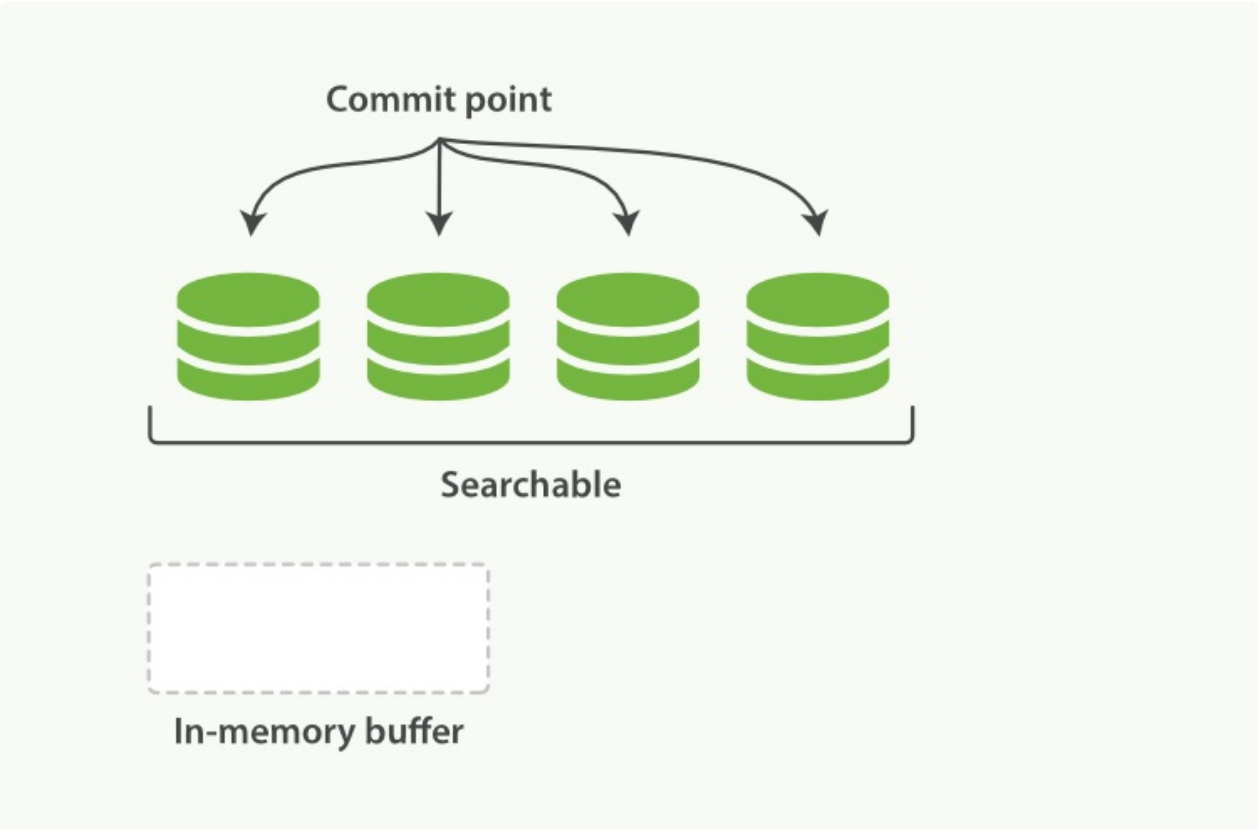


图3：提交后，新的段加到了提交点，缓存被清空



当一个请求被接受，所有段依次查询。所有段上的Term统计信息被聚合，确保每个term和文档的相关性被正确计算。通过这种方式，新的文档以较小的代价加入索引。

删除和更新

段是不可变的，所以文档既不能从旧的段中移除，旧的段也不能更新以反映文档最新的版本。相反，每一个提交点包括一

个.del文件，包含了段上已经被删除的文档。

当一个文档被删除，它实际上只是在.del文件中被标记为删除，依然可以匹配查询，但是最终返回之前会被从结果中删除。

文档的更新操作是类似的：当一个文档被更新，旧版本的文档被标记为删除，新版本的文档在新的段中索引。也许该文档的不同版本都会匹配一个查询，但是更老版本会从结果中删除。

在合并段这节，我们会展示删除的文件是如何从文件系统中清除的。

近实时搜索

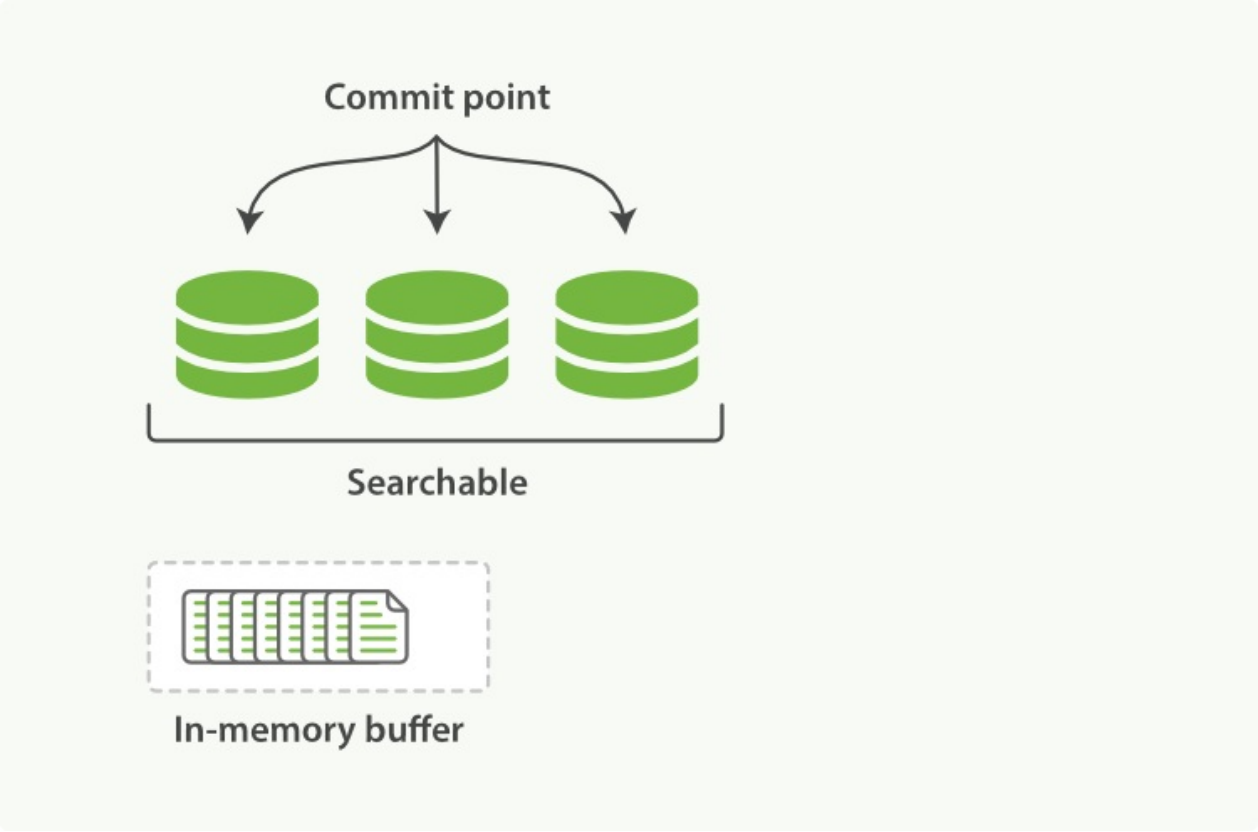
因为 `per-segment search` 机制，索引和搜索一个文档之间是有延迟的。新的文档会在几分钟内可以搜索，但是这依然不够快。

磁盘是瓶颈。提交一个新的段到磁盘需要 `fsync` 操作，确保段被物理地写入磁盘，即时电源失效也不会丢失数据。但是 `fsync` 是昂贵的，它不能在每个文档被索引的时就触发。

所以需要一种更轻量级的方式使新的文档可以被搜索，这意味这移除 `fsync` 。

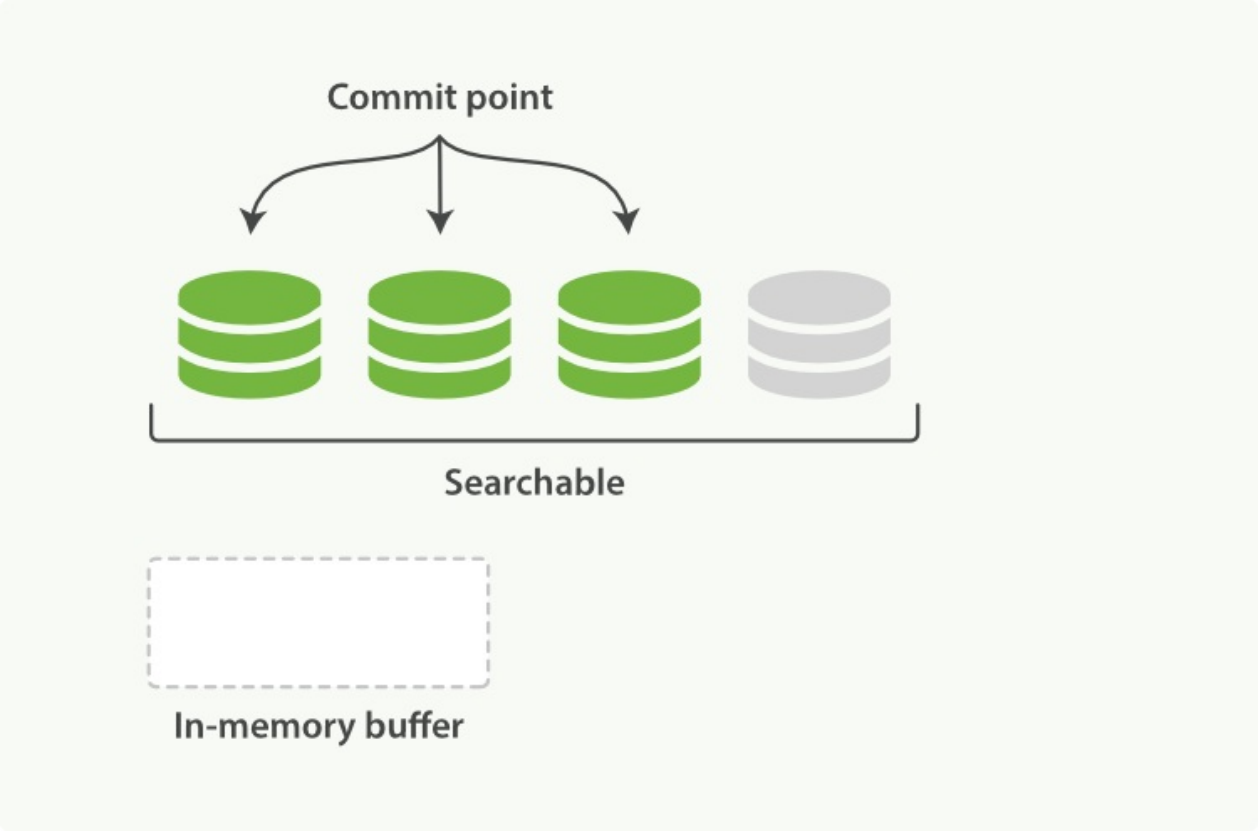
位于Elasticsearch和磁盘间的是文件系统缓存。如前所说，在内存索引缓存中的文档（图1）被写入新的段（图2），但是新的段首先写入文件系统缓存，这代价很低，之后会被同步到磁盘，这个代价很大。但是一旦一个文件被缓存，它也可以被打开和读取，就像其他文件一样。

图**1**：内存缓存区有新文档的**Lucene**索引



Lucene允许新段写入打开，好让它们包括的文档可搜索，而不用执行一次全量提交。这是比提交更轻量的过程，可以经常操作，而不会影响性能。

图**2**：缓存内容已经写到段中，但是还没提交



refeash API

在Elesticsearch中，这种写入打开一个新段的轻量级过程，叫做refresh。默认情况下，每个分片每秒自动刷新一次。这就是为什么说Elasticsearch是近实时的搜索了：文档的改动不会立即被搜索，但是会在一秒内可见。

这会困扰新用户：他们索引了个文档，尝试搜索它，但是搜不到。解决办法就是执行一次手动刷新，通过API:

```
POST /_refresh <1>
POST /blogs/_refresh <2>
```

- <1> refresh所有索引
- <2> 只refresh 索引 `blogs`

> 虽然刷新比提交更轻量，但是它依然有消耗。人工刷新在测试写的时有用，但是不要在生产环境中每写一次就执行刷新，这会影响性能。相反，你的应用需要意识到ES近实时搜索的本质，并且容忍它。

不是所有的用户都需要每秒刷新一次。也许你使用ES索引百万日志文件，你更想要优化索引的速度，而不是进实时搜索。你可以通过修改配置项 `refresh_interval` 减少刷新的频率：

```
PUT /my_logs
{
  "settings": {
    "refresh_interval": "30s" <1>
  }
}
```

- <1> 每30s refresh一次 `my_logs`

`refresh_interval` 可以在存在的索引上动态更新。你在创建大索引的时候可以关闭自动刷新，在要使用索引的时候再打开它。

```
PUT /my_logs/_settings
{ "refresh_interval": -1 } <1>

PUT /my_logs/_settings
{ "refresh_interval": "1s" } <2>
```

- <1> 禁用所有自动refresh
- <2> 每秒自动refresh

持久化变更

没用`fsync`同步文件系统缓存到磁盘，我们不能确保电源失效，甚至正常退出应用后，数据的安全。为了ES的可靠性，需要确保变更持久化到磁盘。

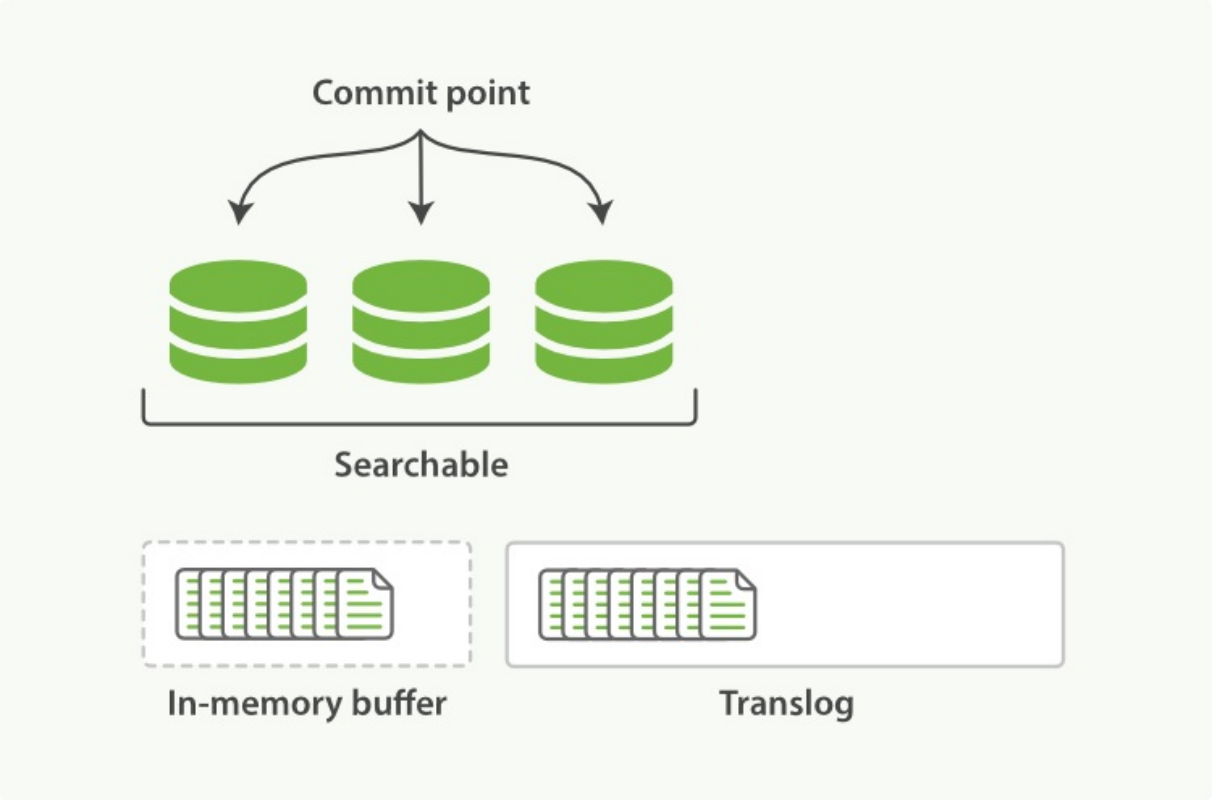
我们说过一次全提交同步段到磁盘，写提交点，这会列出所有的已知的段。在重启，或重新打开索引时，ES使用这次提交点决定哪些段属于当前的分片。

当我们通过每秒的刷新获得近实时的搜索，我们依然需要定时地执行全提交确保能从失败中恢复。但是提交之间的文档怎么办？我们也不想丢失它们。

ES增加了事务日志（`translog`），来记录每次操作。有了事务日志，过程现在如下：

1. 当一个文档被索引，它被加入到内存缓存，同时加到事务日志。

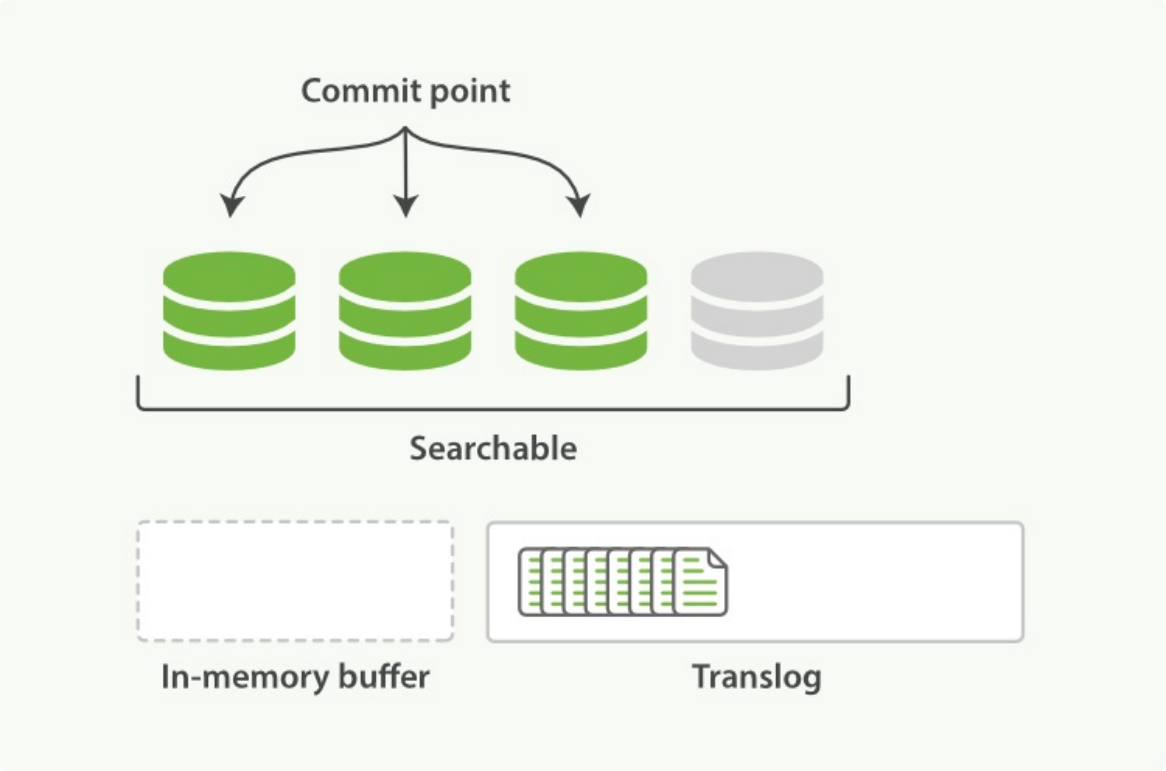
   图**1**：新的文档加入到内存缓存，同时写入事务日志



2. refresh使得分片的进入如下图描述的状态。每秒分片都进行refeash：

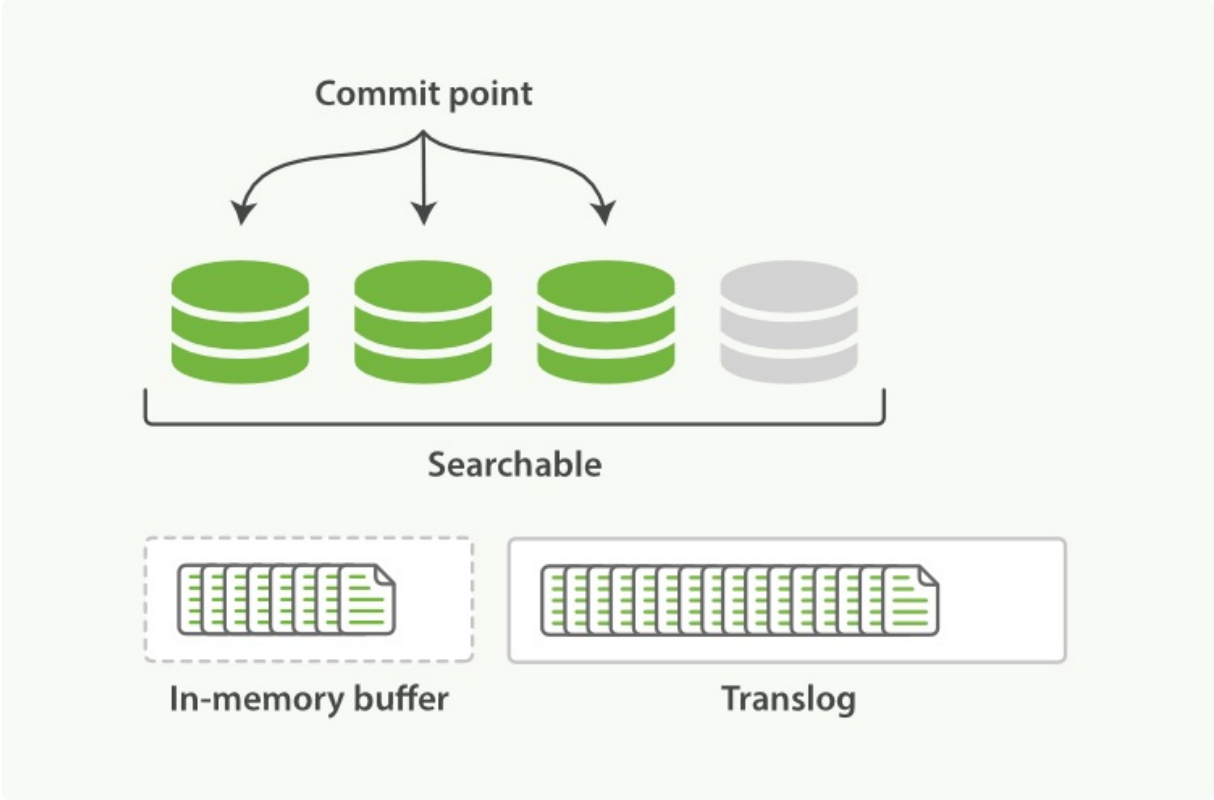
   - 内存缓冲区的文档写入到段中，但没有fsync。
   - 段被打开，使得新的文档可以搜索。
   - 缓存被清除

     图**2**：经过一次**refresh**，缓存被清除，但事务日志没有



3. 随着更多的文档加入到缓存区，写入日志，这个过程会继续

图**3**：事务日志会记录增长的文档



4. 不时地，比如日志很大了，新的日志会创建，会进行一次全提交：

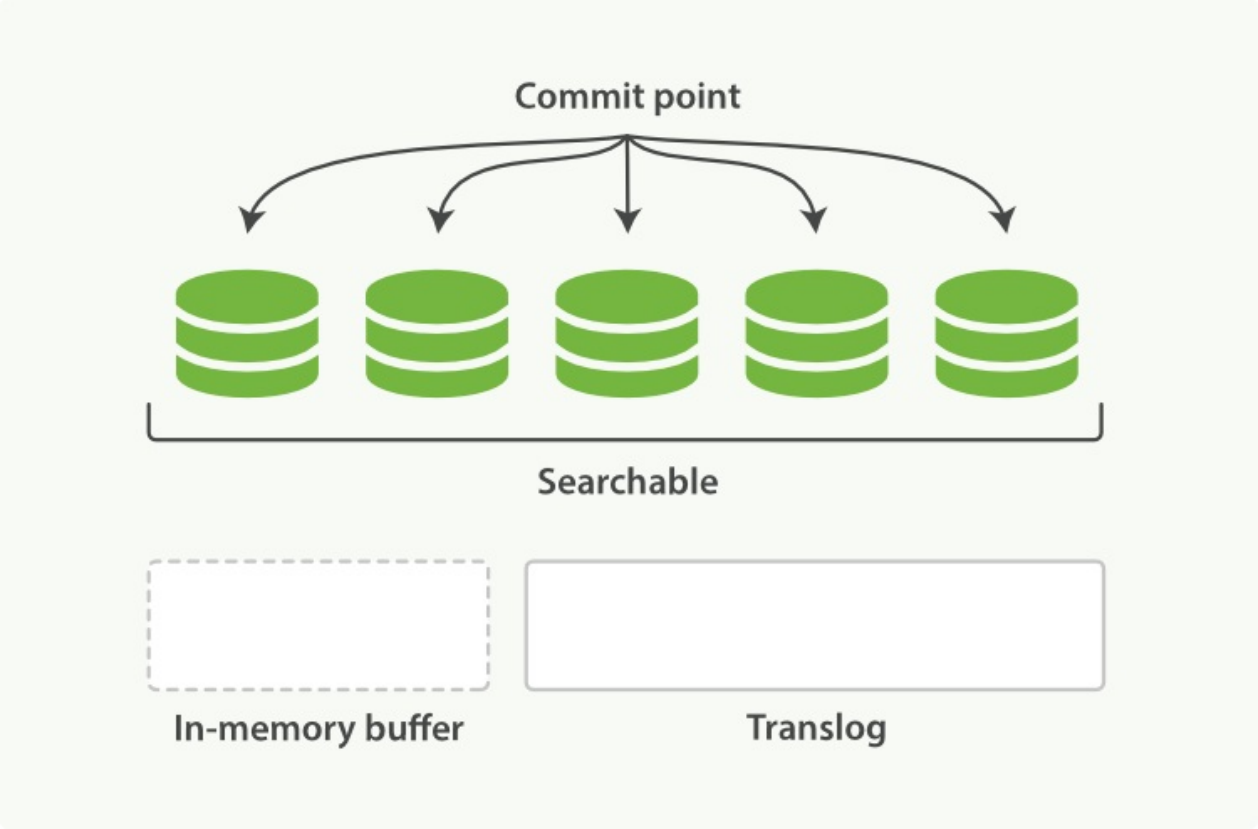
    - 内存缓存区的所有文档会写入到新段中。
    - 清除缓存
    - 一个提交点写入硬盘
    - 文件系统缓存通过fsync操作flush到硬盘
    - 事务日志被清除

事务日志记录了没有flush到硬盘的所有操作。当故障重启后，ES会用最近一次提交点从硬盘恢复所有已知的段，并且从日志

里恢复所有的操作。

事务日志还用来提供实时的CRUD操作。当年用ID进行CRUD时，它在检索相关段内的文档前会首先检查日志最新的改动。这意味着ES可以实时地获取文档的最新版本。

图**4**：**flush**过后，段被全提交，事务日志清除



flush API

在ES中，进行一次提交并删除事务日志的操作叫做 `flush` 。分片每30分钟，或事务日志过大会进行一次flush操作。

`flush API` 可用来进行一次手动flush：

```
POST /blogs/_flush <1>

POST /_flush?wait_for_ongoing  <2>
```

- <1> flush索引 `blogs`
- <2> flush所有索引，等待操作结束再返回

你很少需要手动 `flush` ，通常自动的就够了。

当你要重启或关闭一个索引，flush该索引是很有用的。当ES尝试恢复或者重新打开一个索引时，它必须重放所有事务日志中的操作，所以日志越小，恢复速度越快。

合并段

通过每秒自动刷新创建新的段，用不了多久段的数量就爆炸了。有太多的段是一个问题。每个段消费文件句柄，内存，cpu资源。更重要的是，每次搜索请求都需要依次检查每个段。段越多，查询越慢。

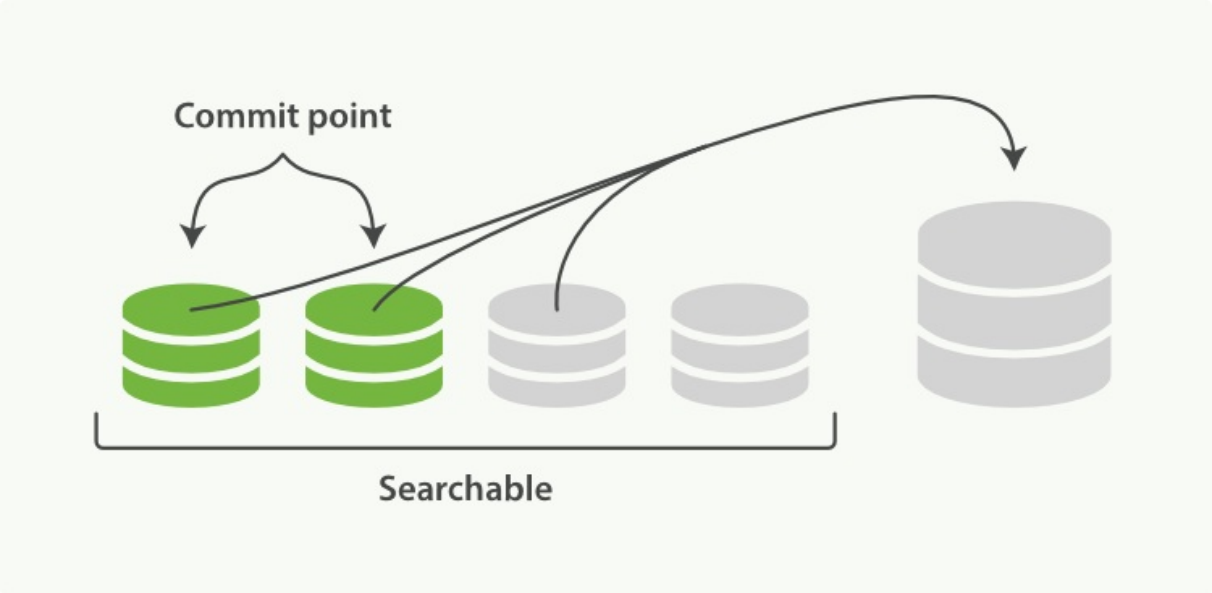
ES通过后台合并段解决这个问题。小段被合并成大段，再合并成更大的段。

这是旧的文档从文件系统删除的时候。旧的段不会再复制到更大的新段中。

这个过程你不必做什么。当你在索引和搜索时ES会自动处理。这个过程如图：两个提交的段和一个未提交的段合并为了一个更大的段所示：

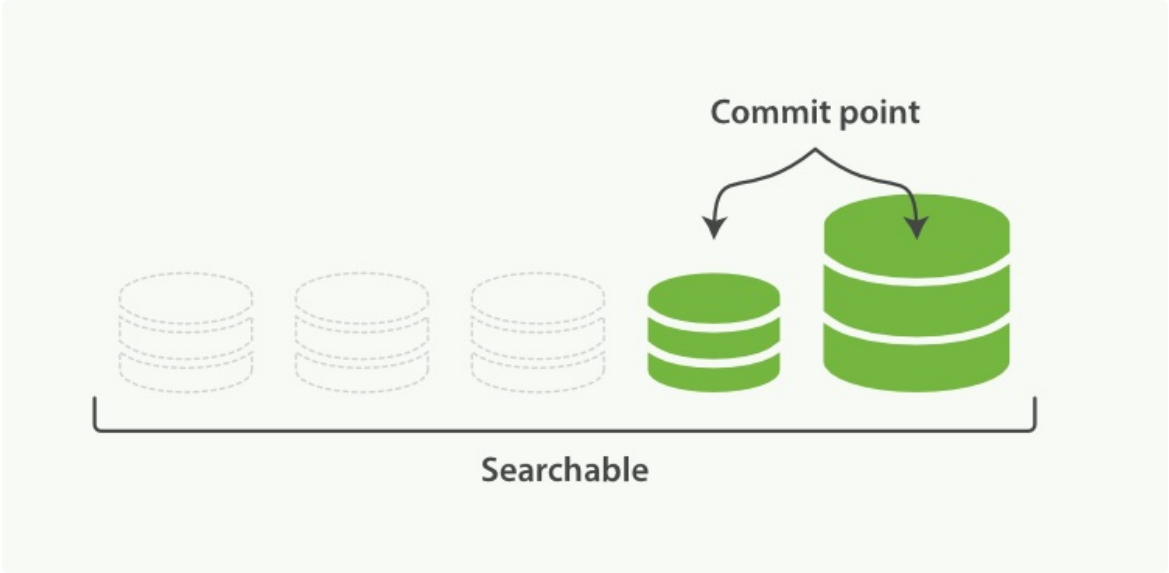
1. 索引过程中，refresh会创建新的段，并打开它。
2. 合并过程会在后台选择一些小的段合并成大的段，这个过程不会中断索引和搜索。

   图**1**：两个提交的段和一个未提交的段合并为了一个更大的段



3. 下图描述了合并后的操作：
   - 新的段flush到了硬盘。
   - 新的提交点写入新的段，排除旧的段。
   - 新的段打开供搜索。
   - 旧的段被删除。

     图**2**：段合并完后，旧的段被删除



合并大的段会消耗很多IO和CPU，如果不检查会影响到搜索性能。默认情况下，ES会限制合并过程，这样搜索就可以有足够的资源进行。

optimize API

`opttimize API` 最好描述为强制合并段API。它强制分片合并段以达到指定 `max_num_segments` 参数。这是为了减少段的数量（通常为1）达到提高搜索性能的目的。

> 警告
>
> 不要在动态的索引（正在活跃更新）上使用 `opttimize API` 。后台的合并处理已经做的很好了，优化命令会阻碍它的工作。不要干涉！

在特定的环境下， `opttimize API` 是有用的。典型的场景是记录日志，这中情况下日志是按照每天，周，月存入索引。旧的索引一般是只可读的，它们是不可能修改的。 这种情况下，把每个索引的段降至1是有效的。搜索过程就会用到更少的资源，性能更好:

```
POST /logstash-2014-10/_optimize?max_num_segments=1 <1>
```

- <1> 把索引中的每个分片都合并成一个段

结构化搜索

结构化搜索 是指查询包含内部结构的数据。日期，时间，和数字都是结构化的：它们有明确的格式给你执行逻辑操作。一般包括比较数字或日期的范围，或确定两个值哪个大。

文本也可以被结构化。一包蜡笔有不同的颜色：`红色`， `绿色`， `蓝色`。一篇博客可能被打上 `分布式` 和 `搜索` 的标签。电子商务产品有商品统一代码（UPCs） 或其他有着严格格式的标识。

通过结构化搜索，你的查询结果始终是 是或非；是否应该属于集合。结构化搜索不关心文档的相关性或分数，它只是简单的包含或排除文档。

这必须是有意义的逻辑，一个数字不能比同一个范围中的其他数字 更多。它只能包含在一个范围中 —— 或不在其中。类似的，对于结构化文本，一个值必须相等或不等。这里没有 更匹配 的概念。

查找准确值

对于准确值，你需要使用过滤器。过滤器的重要性在于它们非常的快。它们不计算相关性（避过所有计分阶段）而且很容易被缓存。我们今后再来讨论过滤器的性能优势【过滤器缓存】，现在，请先记住尽可能多的使用过滤器。

用于数字的 `term` 过滤器

我们下面将介绍 `term` 过滤器，首先因为你可能经常会用到它，这个过滤器旨在处理数字，布尔值，日期，和文本。

我们来看一下例子，一些产品最初用数字来索引，包含两个字段 `price` 和 `productID`：

```
POST /my_store/products/_bulk
{ "index": { "_id": 1 }}
{ "price" : 10, "productID" : "XHDK-A-1293-#fJ3" }
{ "index": { "_id": 2 }}
{ "price" : 20, "productID" : "KDKE-B-9947-#kL5" }
{ "index": { "_id": 3 }}
{ "price" : 30, "productID" : "JODL-X-1937-#pV7" }
{ "index": { "_id": 4 }}
{ "price" : 30, "productID" : "QQPX-R-3956-#aD8" }
```

我们的目标是找出特定价格的产品。假如你有关系型数据库背景，可能用 SQL 来表现这次查询比较熟悉，它看起来像这样：

```
SELECT document
FROM   products
WHERE  price = 20
```

在 Elasticsearch DSL 中，我们使用 `term` 过滤器来实现同样的事。`term` 过滤器会查找我们设定的准确值。`term` 过滤器本身很简单，它接受一个字段名和我们希望查找的值：

```
{
    "term" : {
        "price" : 20
    }
}
```

`term` 过滤器本身并不能起作用。像在【查询 DSL】中介绍的一样，搜索 API 需要得到一个 `查询语句`，而不是一个 `过滤器`。为了使用 `term` 过滤器，我们需要将它包含在一个过滤查询语句中：

```
GET /my_store/products/_search
{
    "query" : {
        "filtered" : { <1>
            "query" : {
                "match_all" : {} <2>
            },
            "filter" : {
                "term" : { <3>
                    "price" : 20
                }
            }
        }
    }
}
```

<1> `filtered` 查询同时接受接受 `query` 与 `filter`。

<2> `match_all` 用来匹配所有文档，这是默认行为，所以在以后的例子中我们将省略掉 `query` 部分。

<3> 这是我们上面见过的 `term` 过滤器。注意它在 `filter` 分句中的位置。

执行之后，你将得到预期的搜索结果：只能文档 2 被返回了（因为只有 `2` 的价格是 `20`）：

```
"hits" : [
    {
        "_index" : "my_store",
        "_type" :  "products",
        "_id" :    "2",
        "_score" : 1.0, <1>
        "_source" : {
          "price" :     20,
          "productID" : "KDKE-B-9947-#kL5"
        }
    }
]
```

<1> 过滤器不会执行计分和计算相关性。分值由 `match_all` 查询产生，所有文档一视同仁，所有每个结果的分值都是 `1`

用于文本的 `term` 过滤器

像我们在开头提到的，`term` 过滤器可以像匹配数字一样轻松的匹配字符串。让我们通过特定 UPC 标识码来找出产品，而不是通过价格。如果用 SQL 来实现，我们可能会使用下面的查询：

```
SELECT product
FROM   products
WHERE  productID = "XHDK-A-1293-#fJ3"
```

转到查询 DSL，我们用 `term` 过滤器来构造一个类似的查询：

```
GET /my_store/products/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "term" : {
                    "productID" : "XHDK-A-1293-#fJ3"
                }
            }
        }
    }
}
```

有点出乎意料：我们没有得到任何结果值！为什么呢？问题不在于 `term` 查询；而在于数据被索引的方式。如果我们使用 `analyze` API，我们可以看到 UPC 被分解成短小的表征：

```
GET /my_store/_analyze?field=productID
XHDK-A-1293-#fJ3
```

```
{
  "tokens" : [ {
    "token" :        "xhdk",
    "start_offset" : 0,
    "end_offset" :   4,
    "type" :         "<ALPHANUM>",
    "position" :     1
  }, {
    "token" :        "a",
    "start_offset" : 5,
    "end_offset" :   6,
    "type" :         "<ALPHANUM>",
    "position" :     2
  }, {
    "token" :        "1293",
    "start_offset" : 7,
    "end_offset" :   11,
```

```
    "type" :         "<NUM>",
    "position" :     3
  }, {
    "token" :        "fj3",
    "start_offset" : 13,
    "end_offset" :   16,
    "type" :         "<ALPHANUM>",
    "position" :     4
  } ]
}
```

这里有一些要点：

- 我们得到了四个分开的表征，而不是一个完整的表征来表示 UPC。
- 所有的字符都被转为了小写。
- 我们失去了连字符和 `#` 符号。

所以当我们用 `XHDK-A-1293-#fJ3` 来查找时，得不到任何结果，因为这个表征不在我们的倒排索引中。相反，那里有上面列出的四个表征。

显然，在处理唯一标识码，或其他枚举值时，这不是我们想要的结果。

为了避免这种情况发生，我们需要通过设置这个字段为 `not_analyzed` 来告诉 Elasticsearch 它包含一个准确值。我们曾在【自定义字段映射】中见过它。为了实现目标，我们要先删除旧索引（因为它包含了错误的映射），并创建一个正确映射的索引：

```
DELETE /my_store <1>

PUT /my_store <2>
{
    "mappings" : {
        "products" : {
            "properties" : {
                "productID" : {
                    "type" : "string",
                    "index" : "not_analyzed" <3>
                }
            }
        }
    }

}
```

<1> 必须首先删除索引，因为我们不能修改已经存在的映射。

<2> 删除后，我们可以用自定义的映射来创建它。

<3> 这里我们明确表示不希望 `productID` 被分析。

现在我们可以继续重新索引文档：

```
POST /my_store/products/_bulk
{ "index": { "_id": 1 }}
{ "price" : 10, "productID" : "XHDK-A-1293-#fJ3" }
{ "index": { "_id": 2 }}
{ "price" : 20, "productID" : "KDKE-B-9947-#kL5" }
{ "index": { "_id": 3 }}
{ "price" : 30, "productID" : "JODL-X-1937-#pV7" }
{ "index": { "_id": 4 }}
{ "price" : 30, "productID" : "QQPX-R-3956-#aD8" }
```

现在我们的 `term` 过滤器将按预期工作。让我们在新索引的数据上再试一次（注意，查询和过滤都没有修改，只是数据被重新映射了）。

```
GET /my_store/products/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "term" : {
                    "productID" : "XHDK-A-1293-#fJ3"
                }
            }
        }
    }
}
```

`productID` 字段没有经过分析， `term` 过滤器也没有执行分析，所以这条查询找到了准确匹配的值，如期返回了文档 1。

内部过滤操作

Elasticsearch 在内部会通过一些操作来执行一次过滤：

1. 查找匹配文档。

   `term` 过滤器在倒排索引中查找词 `XHDK-A-1293-#fJ3`，然后返回包含那个词的文档列表。在这个例子中，只有文档 1 有我们想要的词。

2. 创建字节集

   然后过滤器将创建一个 字节集 —— 一个由 1 和 0 组成的数组 —— 描述哪些文档包含这个词。匹配的文档得到 `1` 字节，在我们的例子中，字节集将是 `[1,0,0,0]`

3. 缓存字节集

   最后，字节集被储存在内存中，以使我们能用它来跳过步骤 1 和 2。这大大的提升了性能，让过滤变得非常的快。

当执行 `filtered` 查询时， `filter` 会比 `query` 早执行。结果字节集会被传给 `query` 来跳过已经被排除的文档。这种过滤器提升性能的方式，查询更少的文档意味着更快的速度。

组合过滤

前面的两个例子展示了单个过滤器的使用。现实中，你可能需要过滤多个值或字段，例如，想在 Elasticsearch 中表达这句 SQL 吗？

```
SELECT product
FROM   products
WHERE  (price = 20 OR productID = "XHDK-A-1293-#fJ3")
  AND  (price != 30)
```

这些情况下，你需要 `bool` 过滤器。这是以其他过滤器作为参数的组合过滤器，将它们结合成多种布尔组合。

布尔过滤器

`bool` 过滤器由三部分组成：

```
{
   "bool" : {
      "must" :     [],
      "should" :   [],
      "must_not" : [],
   }
}
```

`must` ：所有分句都必须匹配，与 `AND` 相同。

`must_not` ：所有分句都必须不匹配，与 `NOT` 相同。

`should` ：至少有一个分句匹配，与 `OR` 相同。

这样就行了！假如你需要多个过滤器，将他们放入 `bool` 过滤器就行。

提示： `bool` 过滤器的每个部分都是可选的（例如，你可以只保留一个 `must` 分句），而且每个部分可以包含一到多个过滤器

为了复制上面的 SQL 示例，我们将两个 `term` 过滤器放在 `bool` 过滤器的 `should` 分句下，然后用另一个分句来处理 `NOT` 条件：

```
GET /my_store/products/_search
{
   "query" : {
      "filtered" : { <1>
         "filter" : {
            "bool" : {
              "should" : [
                 { "term" : {"price" : 20}}, <2>
                 { "term" : {"productID" : "XHDK-A-1293-#fJ3"}} <2>
              ],
              "must_not" : {
                 "term" : {"price" : 30} <3>
              }
           }
         }
      }
   }
}
```

<1> 注意我们仍然需要用 `filtered` 查询来包裹所有条件。

<2> 这两个 `term` 过滤器是 `bool` 过滤器的子节点，因为它们被放在 `should` 分句下，所以至少他们要有一个条件符合。

<3> 如果一个产品价值 `30`，它就会被自动排除掉，因为它匹配了 `must_not` 分句。

我们的搜索结果返回了两个结果，分别满足了 `bool` 过滤器中的不同分句：

```
"hits" : [
    {
        "_id" :     "1",
        "_score" :  1.0,
        "_source" : {
          "price" :     10,
          "productID" : "XHDK-A-1293-#fJ3" <1>
        }
    },
    {
        "_id" :     "2",
        "_score" :  1.0,
        "_source" : {
          "price" :     20, <2>
          "productID" : "KDKE-B-9947-#kL5"
        }
    }
]
```

<1> 匹配 `term` 过滤器 `productID = "XHDK-A-1293-#fJ3"`

<2> 匹配 `term` 过滤器 `price = 20`

嵌套布尔过滤器

虽然 `bool` 是一个组合过滤器而且接受子过滤器，需明白它自己仍然只是一个过滤器。这意味着你可以在 `bool` 过滤器中嵌套 `bool` 过滤器，让你实现更复杂的布尔逻辑。

下面先给出 SQL 语句：

```
SELECT document
FROM   products
WHERE  productID      = "KDKE-B-9947-#kL5"
  OR (     productID = "JODL-X-1937-#pV7"
       AND price     = 30 )
```

我们可以将它翻译成一对嵌套的 `bool` 过滤器：

```
GET /my_store/products/_search
{
   "query" : {
      "filtered" : {
         "filter" : {
            "bool" : {
              "should" : [
                { "term" : {"productID" : "KDKE-B-9947-#kL5"}}, <1>
                { "bool" : { <1>
                  "must" : [
                    { "term" : {"productID" : "JODL-X-1937-#pV7"}}, <2>
                    { "term" : {"price" : 30}} <2>
                  ]
                }}
              ]
            }
         }
      }
   }
}
```

<1> 因为 `term` 和 `bool` 在第一个 `should` 分句中是平级的，至少需要匹配其中的一个过滤器。

<2> `must` 分句中有两个平级的 `term` 分句，所以他们俩都需要匹配。

结果得到两个文档，分别匹配一个 `should` 分句：

```
"hits" : [
    {
        "_id" :     "2",
        "_score" :  1.0,
        "_source" : {
          "price" :     20,
          "productID" : "KDKE-B-9947-#kL5" <1>
        }
    },
    {
        "_id" :     "3",
        "_score" :  1.0,
        "_source" : {
          "price" :      30, <2>
          "productID" : "JODL-X-1937-#pV7" <2>
        }
    }
]
```

<1> `productID` 匹配第一个 `bool` 中的 `term` 过滤器。

<2> 这两个字段匹配嵌套的 `bool` 中的 `term` 过滤器。

这只是一个简单的例子，但是它展示了该怎样用布尔过滤器来构造复杂的逻辑条件。

查询多个准确值

`term` 过滤器在查询单个值时很好用，但是你可能经常需要搜索多个值。比如你想寻找 20 或 30 元的文档，该怎么做呢？

比起使用多个 `term` 过滤器，你可以用一个 `terms` 过滤器。`terms` 过滤器是 `term` 过滤器的复数版本。

它用起来和 `term` 差不多，我们现在来指定一组数值，而不是单一价格：

```
{
    "terms" : {
        "price" : [20, 30]
    }
}
```

像 `term` 过滤器一样，我们将它放在 `filtered` 查询中：

```
GET /my_store/products/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "terms" : { <1>
                    "price" : [20, 30]
                }
            }
        }
    }
}
```

<1> 这是前面提到的 `terms` 过滤器，放置在 `filtered` 查询中

这条查询将返回第二，第三和第四个文档：

```
"hits" : [
    {
        "_id" :    "2",
        "_score" : 1.0,
        "_source" : {
          "price" :     20,
          "productID" : "KDKE-B-9947-#kL5"
        }
    },
    {
        "_id" :    "3",
        "_score" : 1.0,
        "_source" : {
          "price" :     30,
          "productID" : "JODL-X-1937-#pV7"
        }
    },
    {
        "_id":     "4",
        "_score":  1.0,
        "_source": {
           "price":     30,
           "productID": "QQPX-R-3956-#aD8"
        }
     }
]
```

包含，而不是相等

理解 `term` 和 `terms` 是包含操作，而不是相等操作，这点非常重要。这意味着什么？

假如你有一个 term 过滤器 `{ "term" : { "tags" : "search" } }`，它将匹配下面两个文档：

```
{ "tags" : ["search"] }
{ "tags" : ["search", "open_source"] } <1>
```

<1> 虽然这个文档除了 `search` 还有其他短语，它还是被返回了

回顾一下 `term` 过滤器是怎么工作的：它检查倒排索引中所有具有短语的文档，然后组成一个字节集。在我们简单的示例中，我们有下面的倒排索引：

Token	DocIDs
`open_source`	`2`
`search`	`1，2`

当执行 `term` 过滤器来查询 `search` 时，它直接在倒排索引中匹配值并找出相关的 ID。如你所见，文档 1 和文档 2 都包含 `search`，所以他们都作为结果集返回。

提示： 倒排索引的特性让完全匹配一个字段变得非常困难。你将如何确定一个文档只能包含你请求的短语？你将在索引中找出这个短语，解出所有相关文档 ID，然后扫描 索引中每一行来确定文档是否包含其他值。

由此可见，这将变得非常低效和开销巨大。因此， `term` 和 `terms` 是 必须包含 操作，而不是 必须相等。

完全匹配

假如你真的需要完全匹配这种行为，最好是通过添加另一个字段来实现。在这个字段中，你索引原字段包含值的个数。引用上面的两个文档，我们现在包含一个字段来记录标签的个数：

```
{ "tags" : ["search"], "tag_count" : 1 }
{ "tags" : ["search", "open_source"], "tag_count" : 2 }
```

一旦你索引了标签个数，你可以构造一个 `bool` 过滤器来限制短语个数：

```
GET /my_index/my_type/_search
{
    "query": {
        "filtered" : {
            "filter" : {
                 "bool" : {
                    "must" : [
                        { "term" : { "tags" : "search" } }, <1>
                        { "term" : { "tag_count" : 1 } } <2>
                    ]
                }
            }
        }
    }
}
```

<1> 找出所有包含 `search` 短语的文档

<2> 但是确保文档只有一个标签

这将匹配只有一个 `search` 标签的文档，而不是匹配所有包含了 `search` 标签的文档。

范围

我们到现在只搜索过准确的数字，现实中，通过范围来过滤更为有用。例如，你可能希望找到所有价格高于 20 元而低于 40 元的产品。

在 SQL 语法中，范围可以如下表示：

```
SELECT document
FROM   products
WHERE  price BETWEEN 20 AND 40
```

Elasticsearch 有一个 `range` 过滤器，让你可以根据范围过滤：

```
"range" : {
    "price" : {
        "gt" : 20,
        "lt" : 40
    }
}
```

`range` 过滤器既能包含也能排除范围，通过下面的选项：

- `gt`：`>` 大于
- `lt`：`<` 小于
- `gte`：`>=` 大于或等于
- `lte`：`<=` 小于或等于

下面是范围过滤器的一个示例：

```
GET /my_store/products/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "range" : {
                    "price" : {
                        "gte" : 20,
                        "lt"  : 40
                    }
                }
            }
        }
    }
}
```

假如你需要不设限的范围，去掉一边的限制就可以了：

```
"range" : {
    "price" : {
        "gt" : 20
    }
}
```

日期范围

`range` 过滤器也可以用于日期字段：

```
"range" : {
    "timestamp" : {
```

```
        "gt" : "2014-01-01 00:00:00",
        "lt" : "2014-01-07 00:00:00"
    }
}
```

当用于日期字段时，`range` 过滤器支持日期数学操作。例如，我们想找到所有最近一个小时的文档：

```
"range" : {
    "timestamp" : {
        "gt" : "now-1h"
    }
}
```

这个过滤器将始终能找出所有时间戳大于当前时间减 1 小时的文档，让这个过滤器像移窗一样通过你的文档。

日期计算也能用于实际的日期，而不是仅仅是一个像 now 一样的占位符。只要在日期后加上双竖线 `||`，就能使用日期数学表达式了。

```
"range" : {
    "timestamp" : {
        "gt" : "2014-01-01 00:00:00",
        "lt" : "2014-01-01 00:00:00||+1M" <1>
    }
}
```

<1> 早于 2014 年 1 月 1 号加一个月

日期计算是与日历相关的，所以它知道每个月的天数，每年的天数，等等。更详细的关于日期的信息可以在这里找到 日期格式手册

字符串范围

`range` 过滤器也可以用于字符串。字符串范围根据字典或字母顺序来计算。例如，这些值按照字典顺序排序：

- 5, 50, 6, B, C, a, ab, abb, abc, b

提示：倒排索引中的短语按照字典顺序排序，也是为什么字符串范围使用这个顺序。

假如我们想让范围从 `a` 开始而不包含 `b`，我们可以用类似的 `range` 过滤器语法：

```
"range" : {
    "title" : {
        "gte" : "a",
        "lt" :  "b"
    }
}
```

当心基数：

数字和日期字段的索引方式让他们在计算范围时十分高效。但对于字符串来说却不是这样。为了在字符串上执行范围操作，Elasticsearch 会在这个范围内的每个短语执行 `term` 操作。这比日期或数字的范围操作慢得多。

字符串范围适用于一个基数较小的字段，一个唯一短语个数较少的字段。你的唯一短语数越多，搜索就越慢。

处理 Null 值

回到我们早期的示例，在文档中有一个多值的字段 `tags`，一个文档可能包含一个或多个标签，或根本没有标签。如果一个字段没有值，它是怎么储存在倒排索引中的？

这是一个取巧的问题，因为答案是它根本没有存储。让我们从看一下前几节的倒排索引：

Token	DocIDs
`open_source`	`2`
`search`	`1，2`

你怎么可能储存一个在数据结构不存在的字段呢？倒排索引是表征和包含它们的文档的一个简单列表。假如一个字段不存在，它就没有任何表征，也就意味着它无法被倒排索引的数据结构表达出来。

本质上来说，`null`，`[]`（空数组）和 `[null]` 是相等的。它们都不存在于倒排索引中！

显然，这个世界却没有那么简单，数据经常会缺失字段，或包含空值或空数组。为了应对这些情形，Elasticsearch 有一些工具来处理空值或缺失的字段。

`exists` 过滤器

工具箱中的第一个利器是 `exists` 过滤器，这个过滤器将返回任何包含这个字段的文档，让我们用标签来举例，索引一些示例文档：

```
POST /my_index/posts/_bulk
{ "index": { "_id": "1"              }}
{ "tags" : ["search"]                }  <1>
{ "index": { "_id": "2"              }}
{ "tags" : ["search", "open_source"] }  <2>
{ "index": { "_id": "3"              }}
{ "other_field" : "some data"        }  <3>
{ "index": { "_id": "4"              }}
{ "tags" : null                      }  <4>
{ "index": { "_id": "5"              }}
{ "tags" : ["search", null]          }  <5>
```

<1> `tags` 字段有一个值

<2> `tags` 字段有两个值

<3> `tags` 字段不存在

<4> `tags` 字段被设为 `null`

<5> `tags` 字段有一个值和一个 `null`

结果我们 `tags` 字段的倒排索引看起来将是这样：

Token	DocIDs
`open_source`	`2`
`search`	`1，2，5`

我们的目标是找出所有设置了标签的文档，我们不关心这个标签是什么，只要它存在于文档中就行。在 SQL 语法中，我们可以用 `IS NOT NULL` 查询：

```
SELECT tags
FROM   posts
```

```
WHERE  tags IS NOT NULL
```

在 Elasticsearch 中，我们使用 `exists` 过滤器：

```
GET /my_index/posts/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "exists" : { "field" : "tags" }
            }
        }
    }
}
```

查询返回三个文档：

```
"hits" : [
    {
      "_id" :     "1",
      "_score" :  1.0,
      "_source" : { "tags" : ["search"] }
    },
    {
      "_id" :     "5",
      "_score" :  1.0,
      "_source" : { "tags" : ["search", null] } <1>
    },
    {
      "_id" :     "2",
      "_score" :  1.0,
      "_source" : { "tags" : ["search", "open source"] }
    }
]
```

<1> 文档 5 虽然包含了一个 `null` 值，仍被返回了。这个字段存在是因为一个有值的标签被索引了，所以 `null` 对这个过滤器没有影响

结果很容易理解，所以在 `tags` 字段中有值的文档都被返回了。只排除了文档 3 和 4。

`missing` 过滤器

`missing` 过滤器本质上是 `exists` 的反义词：它返回没有特定字段值的文档，像这条 SQL 一样：

```
SELECT tags
FROM   posts
WHERE  tags IS  NULL
```

让我们在前面的例子中用 `missing` 过滤器来取代 `exists` ：

```
GET /my_index/posts/_search
{
    "query" : {
        "filtered" : {
            "filter": {
                "missing" : { "field" : "tags" }
            }
        }
    }
}
```

如你所愿，我们得到了两个没有包含标签字段的文档：

```
"hits" : [
    {
      "_id" :     "3",
      "_score" :  1.0,
      "_source" : { "other_field" : "some data" }
    },
    {
      "_id" :     "4",
      "_score" :  1.0,
      "_source" : { "tags" : null }
    }
]
```

什么时候 null 才表示 null

有时你需要能区分一个字段是没有值，还是被设置为 `null`。用上面见到的默认行为无法区分这一点，数据都不存在了。幸运的是，我们可以将明确的 `null` 值用我们选择的占位符来代替

当指定字符串，数字，布尔值或日期字段的映射时，你可以设置一个 `null_value` 来处理明确的 `null` 值。没有值的字段仍将被排除在倒排索引外。

当选定一个合适的 `null_value` 时，确保以下几点：

- 它与字段的类型匹配，你不能在 `date` 类型的字段中使用字符串 `null_value`
- 它需要能与这个字段可能包含的正常值区分开来，以避免真实值和 `null` 值混淆

对象的 `exists/missing`

`exists` 和 `missing` 过滤器同样能在内联对象上工作，而不仅仅是核心类型。例如下面的文档：

```
{
   "name" : {
      "first" : "John",
      "last" :  "Smith"
   }
}
```

你可以检查 `name.first` 和 `name.last` 的存在性，也可以检查 `name` 的。然而，在【映射】中，我们提到对象在内部被转成扁平化的键值结构，像下面所示：

```
{
   "name.first" : "John",
   "name.last"  : "Smith"
}
```

所以我们是怎么使用 `exists` 或 `missing` 来检测 `name` 字段的呢，这个字段并没有真正存在于倒排索引中。

原因是像这样的一个过滤器

```
{
    "exists" : { "field" : "name" }
}
```

实际是这样执行的

```
{
    "bool": {
        "should": [
            { "exists": { "field": { "name.first" }}},
            { "exists": { "field": { "name.last"  }}}
        ]
```

```
    }
}
```

同样这意味着假如 `first` 和 `last` 都为空，那么 `name` 就是不存在的。

关于缓存

在【内部过滤操作】章节中，我们简单提到过过滤器是怎么计算的。它们的核心是一个字节集来表示哪些文档符合这个过滤器。Elasticsearch 主动缓存了这些字节集留作以后使用。一旦缓存后，当遇到相同的过滤时，这些字节集就可以被重用，而不需要重新运算整个过滤。

缓存的字节集很“聪明”：他们会增量更新。你索引中添加了新的文档，只有这些新文档需要被添加到已存的字节集中，而不是一遍遍重新计算整个缓存的过滤器。过滤器和整个系统的其他部分一样是实时的，你不需要关心缓存的过期时间。

独立的过滤缓存

每个过滤器都被独立计算和缓存，而不管它们在哪里使用。如果两个不同的查询使用相同的过滤器，则会使用相同的字节集。同样，如果一个查询在多处使用同样的过滤器，只有一个字节集会被计算和重用。

让我们看一下示例，查找符合下列条件的邮箱：

- 在收件箱而且没有被读取过
- 不在收件箱但是被标记为重要

```
"bool": {
   "should": [
      { "bool": {
            "must": [
               { "term": { "folder": "inbox" }}, <1>
               { "term": { "read": false }}
            ]
      }},
      { "bool": {
            "must_not": {
               "term": { "folder": "inbox" } <1>
            },
            "must": {
               "term": { "important": true }
            }
      }}
   ]
}
```

<1> 这两个过滤器相同，而且会使用同一个字节集。

虽然一个收件箱条件是 `must` 而另一个是 `must_not`，这两个条件本身是相等的。这意味着字节集会在第一个条件执行时计算一次，然后作为缓存被另一个条件使用。而第二次执行这条查询时，收件箱的过滤已经被缓存了，所以两个条件都能使用缓存的字节集。

这与查询 DSL 的组合型紧密相关。移动过滤器或在相同查询中多处重用相同的过滤器非常简单。这不仅仅是方便了开发者——对于性能也有很大的提升

控制缓存

大部分直接处理字段的枝叶过滤器（例如 `term`）会被缓存，而像 `bool` 这类的组合过滤器则不会被缓存。

【提示】

枝叶过滤器需要在硬盘中检索倒排索引，所以缓存它们是有意义的。另一方面来说，组合过滤器使用快捷的字节逻辑来组合它们内部条件生成的字节集结果，所以每次重新计算它们也是很高效的。

然而，有部分枝叶过滤器，默认不会被缓存，因为它们这样做没有意义：

脚本过滤器：

脚本过滤器的结果不能被缓存因为脚本的意义对于 Elasticsearch 来说是不透明的。

Geo 过滤器：

定位过滤器（我们会在【geoloc】中更详细的介绍），通常被用于过滤基于特定用户地理位置的结果。因为每个用户都有一个唯一的定位，geo 过滤器看起来不太会重用，所以缓存它们没有意义。

日期范围：

使用 `now` 方法的日期范围（例如 `"now-1h"`），结果值精确到毫秒。每次这个过滤器执行时，`now` 返回一个新的值。老的过滤器将不再被使用，所以默认缓存是被禁用的。然而，当 `now` 被取整时（例如，`now/d` 取最近一天），缓存默认是被启用的。

有时候默认的缓存测试并不正确。可能你希望一个复杂的 `bool` 表达式可以在相同的查询中重复使用，或你想要禁用一个 `date` 字段的过滤器缓存。你可以通过 `_cache` 标记来覆盖几乎所有过滤器的默认缓存策略

```
{
    "range" : {
        "timestamp" : {
            "gt" : "2014-01-02 16:15:14" <1>
        },
        "_cache": false <2>
    }
}
```

<1> 看起来我们不会再使用这个精确时间戳

<2> 在这个过滤器上禁用缓存

以后的章节将提供一些例子来说明哪些时候覆盖默认缓存策略是有意义的。

过滤顺序

在 `bool` 条件中过滤器的顺序对性能有很大的影响。更详细的过滤条件应该被放置在其他过滤器之前，以便在更早的排除更多的文档。

假如条件 A 匹配 1000 万个文档，而 B 只匹配 100 个文档，那么需要将 B 放在 A 前面。

缓存的过滤器非常快，所以它们需要被放在不能缓存的过滤器之前。想象一下我们有一个索引包含了一个月的日志事件，然而，我们只对近一个小时的事件感兴趣：

```
GET /logs/2014-01/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "range" : {
                    "timestamp" : {
                        "gt" : "now-1h"
                    }
                }
            }
        }
    }
}
```

这个过滤条件没有被缓存，因为它使用了 `now` 方法，这个值每毫秒都在变化。这意味着我们需要每次执行这条查询时都检测一整个月的日志事件。

我们可以通过组合一个缓存的过滤器来让这变得更有效率：我们可以添加一个含固定时间的过滤器来排除掉这个月的大部分数据，例如昨晚凌晨：

```
"bool": {
    "must": [
        { "range" : {
            "timestamp" : {
                "gt" : "now-1h/d" <1>
            }
        }},
        { "range" : {
            "timestamp" : {
                "gt" : "now-1h" <2>
            }
        }}
    ]
}
```

<1> 这个过滤器被缓存了，因为它使用了取整到昨夜凌晨 `now` 条件。

<2> 这个过滤器没有被缓存，因为它没有对 `now` 取整。

`now-1h/d` 条件取整到昨夜凌晨，所以所有今天之前的文档都被排除掉了。这个结果的字节集被缓存了，因为 `now` 被取整了，意味着它只需要每天当昨夜凌晨的值改变时被执行一次。`now-1h` 条件没有被缓存，因为 `now` 表示最近一毫秒的时间。然而，得益于第一个过滤器，第二个过滤器只需要检测当天的文档就行。

这些条件的排序很重要。上面的实现能正常工作是因为自从昨晚凌晨条件比最近一小时条件位置更前。假如它们用别的方式组合，那么最近一小时条件还是需要检测所有的文档，而不仅仅是昨夜以来的文档。

[[full-text-search]] == Full-Text Search

Now that we have covered the simple case of searching for structured data, it is time to (((“full text search”)))explore *full-text search*: how to search within full-text fields in order to find the most relevant documents.

The two most important aspects of (((“relevance”)))full-text search are as follows:

Relevance::

```
The ability to rank results by how relevant they are to
the given query, whether relevance is calculated using
TF/IDF (see <<relevance-intro>>), proximity to a geolocation,
fuzzy similarity, or some other algorithm.
```

Analysis::

```
The process of converting a block of text into distinct, normalized tokens
(see <<analysis-intro>>) in order to (a) create an inverted index and
(b) query the inverted index.
```

As soon as we talk (((“analysis”)))about either relevance or analysis, we are in the territory of queries, rather than filters.

[[term-vs-full-text]] === Term-Based Versus Full-Text

While all queries perform some sort of relevance calculation, not all queries have an analysis phase.(((“full text search”, “term-based versus”)))(((“term-based queries”))) Besides specialized queries like the `bool` or `function_score` queries, which don't operate on text at all, textual queries can be broken down into two families:

Term-based queries::

+

Queries like the `term` or `fuzzy` queries are low-level queries that have no analysis phase.(((“fuzzy queries”))) They operate on a single term. A `term` query for the term `Foo` looks for that *exact term* in the inverted index and calculates the TF/IDF relevance `_score` for each document that contains the term.

It is important to remember that the `term` query looks in the inverted index for the exact term only; it won't match any variants like `foo` or `FOO`. It doesn't matter how the term came to be in the index, just that it is. If you were to index `["Foo","Bar"]` into an exact value `not_analyzed` field, or `Foo Bar` into an analyzed field with the `whitespace` analyzer, both would result in having the two terms `Foo` and `Bar` in the inverted index.

--

Full-text queries::

+

Queries like the `match` or `query_string` queries are high-level queries that understand the mapping of a field:

- If you use them to query a `date` or `integer` field, they will treat the query string as a date or integer, respectively.
- If you query an exact value (`not_analyzed`) string field,(((“not_analyzed string fields”, “match or query-string queries on”))) they will treat the whole query string as a single term.
- But if you query a full-text (`analyzed`) field,(((“analyzed fields”, “match or query-string queries on”))) they will first pass the query string through the appropriate analyzer to produce the list of terms to be queried.

Once the query has assembled a list of terms, it executes the appropriate low-level query for each of these terms, and then combines their results to produce the final relevance score for each document.

We will discuss this process in more detail in the following chapters.

You seldom need to use the term-based queries directly. Usually you want to query full text, not individual terms, and this is easier to do with the high-level full-text queries (which end up using term-based queries internally).

[NOTE]

If you do find yourself wanting to use a query on an exact value `not_analyzed` field, (("exact values", "not_analyzed fields, querying")))think about whether you really want a query or a filter.

Single-term queries usually represent binary yes/no questions and are almost always better expressed as a (("filters", "single-term queries better expressed as")))filter, so that they can benefit from <>:

[source,js]

GET /_search { "query": { "filtered": { "filter": { "term": { "gender": "female" } } } }

}

====

[[match-query]] === The match Query

The `match` query is the *go-to* query--the first query that you should reach for whenever you need to query any field. ((("match query")))((("full text search", "match query"))) It is a high-level *full-text query*, meaning that it knows how to deal with both full-text fields and exact-value fields.

That said, the main use case for the `match` query is for full-text search. So let's take a look at how full-text search works with a simple example.

[[match-test-data]] ==== Index Some Data

First, we'll create a new index and index some(((("full text search", "match query", "indexing data"))) documents using the <>:

[source,js]

DELETE /my_index <1>

PUT /my_index { "settings": { "number_of_shards": 1 }} <2>

POST /my_index/my_type/_bulk { "index": { "_id": 1 }} { "title": "The quick brown fox" } { "index": { "_id": 2 }} { "title": "The quick brown fox jumps over the lazy dog" } { "index": { "_id": 3 }} { "title": "The quick brown fox jumps over the quick dog" } { "index": { "_id": 4 }}

{ "title": "Brown fox brown dog" }

// SENSE: 100_Full_Text_Search/05_Match_query.json

<1> Delete the index in case it already exists.

<2> Later, in <>, we explain why we created this index with only one primary shard.

==== A Single-Word Query

Our first example explains what(((("full text search", "match query", "single word query")))((("match query", "single word query"))) happens when we use the `match` query to search within a full-text field for a single word:

[source,js]

GET /my_index/my_type/_search { "query": { "match": { "title": "QUICK!" } }

}

// SENSE: 100_Full_Text_Search/05_Match_query.json

Elasticsearch executes the preceding `match` query(((("analysis", "in single term match query"))) as follows:

1. *Check the field type*. + The `title` field is a full-text (`analyzed`) `string` field, which means that the query string should be analyzed too.

2. *Analyze the query string*. + The query string `QUICK!` is passed through the standard analyzer, which results in the single term `quick` . Because we have a just a single term, the `match` query can be executed as a single low-level `term` query.

3. *Find matching docs*. + The `term` query looks up `quick` in the inverted index and retrieves the list of documents that

contain that term--in this case, documents 1, 2, and 3.

4. *Score each doc*. + The `term` query calculates the relevance `_score` for each matching document, by combining the((("relevance scores", "calculating for single term match query results"))) term frequency (how often `quick` appears in the `title` field of each document), with the inverse document frequency (how often `quick` appears in the `title` field in *all* documents in the index), and the length of each field (shorter fields are considered more relevant). See <>.

This process gives us the following (abbreviated) results:

[source,js]

"hits": [{ "_id": "1", "_score": 0.5, <1> "_source": { "title": "The quick brown fox" } }, { "_id": "3", "_score": 0.44194174, <2> "_source": { "title": "The quick brown fox jumps over the quick dog" } }, { "_id": "2", "_score": 0.3125, <2> "_source": { "title": "The quick brown fox jumps over the lazy dog" } }

]

<1> Document 1 is most relevant because its `title` field is short, which means that `quick` represents a large portion of its content.

<2> Document 3 is more relevant than document 2 because `quick` appears twice.

[[match-multi-word]] === Multiword Queries

If we could search for only one word at a time, full-text search would be pretty inflexible. Fortunately, the `match` query(((("full text search", "multi-word queries")))(((("match query", "multi-word query"))) makes multiword queries just as simple:

[source,js]

GET /my_index/my_type/_search { "query": { "match": { "title": "BROWN DOG!" } }

}

// SENSE: 100_Full_Text_Search/05_Match_query.json

The preceding query returns all four documents in the results list:

[source,js]

{ "hits": [{ "_id": "4", "_score": 0.73185337, <1> "_source": { "title": "Brown fox brown dog" } }, { "_id": "2", "_score": 0.47486103, <2> "_source": { "title": "The quick brown fox jumps over the lazy dog" } }, { "_id": "3", "_score": 0.47486103, <2> "_source": { "title": "The quick brown fox jumps over the quick dog" } }, { "_id": "1", "_score": 0.11914785, <3> "_source": { "title": "The quick brown fox" } }]

}

<1> Document 4 is the most relevant because it contains `"brown"` twice and `"dog"` once.

<2> Documents 2 and 3 both contain `brown` and `dog` once each, and the `title` field is the same length in both docs, so they have the same score.

<3> Document 1 matches even though it contains only `brown`, not `dog`.

Because the `match` query has to look for two terms—`["brown","dog"]`—internally it has to execute two `term` queries and combine their individual results into the overall result. To do this, it wraps the two `term` queries in a `bool` query, which we examine in detail in <>.

The important thing to take away from this is that any document whose `title` field contains *at least one of the specified terms* will match the query. The more terms that match, the more relevant the document.

[[match-improving-precision]] ==== Improving Precision

Matching any document that contains *any* of the query terms may result in a long tail of seemingly irrelevant results. ((("full text search", "multi-word queries", "improving precision")))((("precision", "improving for full text search multi-word queries"))) It's a shotgun approach to search. Perhaps we want to show only documents that contain *all* of the query terms. In other words, instead of `brown OR dog`, we want to return only documents that match `brown AND dog`.

The `match` query accepts an `operator` parameter(((("match query", "operator parameter")))((("or operator", "in match queries")))((("and operator", "in match queries"))) that defaults to `or`. You can change it to `and` to require that all specified terms must match:

[source,js]

GET /my_index/my_type/_search { "query": { "match": { "title": { <1> "query": "BROWN DOG!", "operator": "and" } } }

}

// SENSE: 100_Full_Text_Search/05_Match_query.json

<1> The structure of the `match` query has to change slightly in order to accommodate the `operator` parameter.

This query would exclude document 1, which contains only one of the two terms.

[[match-precision]] ==== Controlling Precision

The choice between *all* and *any* is a bit((("full text search", "multi-word queries", "controlling precision"))) too black-or-white. What if the user specified five query terms, and a document contains only four of them? Setting `operator` to `and` would exclude this document.

Sometimes that is exactly what you want, but for most full-text search use cases, you want to include documents that may be relevant but exclude those that are unlikely to be relevant. In other words, we need something in-between.

The `match` query supports((("match query", "minimum_should_match parameter")))(("minimum_should_match parameter"))) the `minimum_should_match` parameter, which allows you to specify the number of terms that must match for a document to be considered relevant. While you can specify an absolute number of terms, it usually makes sense to specify a percentage instead, as you have no control over the number of words the user may enter:

[source,js]

GET /my_index/my_type/_search { "query": { "match": { "title": { "query": "quick brown dog", "minimum_should_match": "75%" } } }

}

// SENSE: 100_Full_Text_Search/05_Match_query.json

When specified as a percentage, `minimum_should_match` does the right thing: in the preceding example with three terms, `75%` would be rounded down to `66.6%`, or two out of the three terms. No matter what you set it to, at least one term must match for a document to be considered a match.

[NOTE]

The `minimum_should_match` parameter is flexible, and different rules can be applied depending on the number of terms the user enters. For the full documentation see the

http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-minimum-should-match.html#query-dsl-minimum-should-match

To fully understand how the `match` query handles multiword queries, we need to look at how to combine multiple queries with the `bool` query.

[[bool-query]] === Combining Queries

In <> we discussed how to(("full text search", "combining queries")), use the `bool` filter to combine multiple filter clauses with `and`, `or`, and `not` logic. In query land, the `bool` query does a similar job but with one important difference.

Filters make a binary decision: should this document be included in the results list or not? Queries, however, are more subtle. They decide not only whether to include a document, but also how *relevant* that document is.

Like the filter equivalent, the `bool` query accepts(("bool query")) multiple query clauses under the `must`, `must_not`, and `should` parameters. For instance:

[source,js]

GET /my_index/my_type/_search { "query": { "bool": { "must": { "match": { "title": "quick" }}, "must_not": { "match": { "title": "lazy" }}, "should": [{ "match": { "title": "brown" }}, { "match": { "title": "dog" }}] } }

}

// SENSE: 100_Full_Text_Search/15_Bool_query.json

The results from the preceding query include any document whose `title` field contains the term `quick`, except for those that also contain `lazy`. So far, this is pretty similar to how the `bool` filter works.

The difference comes in with the two `should` clauses, which say that: a document is *not required* to contain (("should clause", "in bool queries")))either `brown` or `dog`, but if it does, then it should be considered *more relevant*:

[source,js]

{ "hits": [{ "_id": "3", "_score": 0.70134366, <1> "_source": { "title": "The quick brown fox jumps over the quick dog" } }, { "_id": "1", "_score": 0.3312608, "_source": { "title": "The quick brown fox" } }]

}

<1> Document 3 scores higher because it contains both `brown` and `dog`.

==== Score Calculation

The `bool` query calculates(("relevance scores", "calculation in bool queries"))(("bool query", "score calculation")) the relevance `_score` for each document by adding together the `_score` from all of the matching `must` and `should` clauses, and then dividing by the total number of `must` and `should` clauses.

The `must_not` clauses do not affect (("must_not clause", "in bool queries")))the score; their only purpose is to exclude documents that might otherwise have been included.

==== Controlling Precision

All the `must` clauses must match, and all the `must_not` clauses must not match, but how many `should` clauses(("bool query", "controlling precision"))(("full text search", "combining queries", "controlling precision"))(("precision", "controlling for bool query"))) should match? By default, none of the `should` clauses are required to match, with one exception: if there are no `must` clauses, then at least one `should` clause must match.

Just as we can control the <>, we can control how many `should` clauses need to match by using the `minimum_should_match` parameter,(("minimum_should_match parameter", "in bool queries"))) either as an absolute number or as a percentage:

[source,js]

GET /my_index/my_type/_search { "query": { "bool": { "should": [{ "match": { "title": "brown" }}, { "match": { "title": "fox" }}, { "match": { "title": "dog" }}], "minimum_should_match": 2 <1> } }

}

// SENSE: 100_Full_Text_Search/15_Bool_query.json

<1> This could also be expressed as a percentage.

The results would include only documents whose `title` field contains `"brown" AND "fox"`, `"brown" AND "dog"`, or `"fox" AND "dog"`. If a document contains all three, it would be considered more relevant than those that contain just two of the three.

=== How match Uses bool

By now, you have probably realized that <> simply wrap((("match query", "use of bool query in multi-word searches"))) ((("bool query", "use by match query in multi-word searches")))((("full text search", "how match query uses bool query"))) the generated `term` queries in a `bool` query. With the default `or` operator, each `term` query is added as a `should` clause, so at least one clause must match. These two queries are equivalent:

[source,js]

{ "match": { "title": "brown fox"}

}

[source,js]

{ "bool": { "should": [{ "term": { "title": "brown" }}, { "term": { "title": "fox" }}] }

}

With the `and` operator, all the `term` queries are added as `must` clauses, so *all* clauses must match. These two queries are equivalent:

[source,js]

{ "match": { "title": { "query": "brown fox", "operator": "and" } }

}

[source,js]

{ "bool": { "must": [{ "term": { "title": "brown" }}, { "term": { "title": "fox" }}] }

}

And if the `minimum_should_match` parameter is((("minimum_should_match parameter", "match query using bool query"))) specified, it is passed directly through to the `bool` query, making these two queries equivalent:

[source,js]

{ "match": { "title": { "query": "quick brown fox", "minimum_should_match": "75%" } }

}

[source,js]

{ "bool": { "should": [{ "term": { "title": "brown" }}, { "term": { "title": "fox" }}, { "term": { "title": "quick" }}], "minimum_should_match": 2 <1> }

```
}
```

<1> Because there are only three clauses, the `minimum_should_match` value of `75%` in the `match` query is rounded down to `2`. At least two out of the three `should` clauses must match.

Of course, we would normally write these types of queries by using the `match` query, but understanding how the `match` query works internally lets you take control of the process when you need to. Some things can't be done with a single `match` query, such as give more weight to some query terms than to others. We will look at an example of this in the next section.

=== Boosting Query Clauses

Of course, the `bool` query isn't restricted (((("full text search", "boosting query clauses")))to combining simple one-word `match` queries. It can combine any other query, including other `bool` queries.((("relevance scores", "controlling weight of query clauses"))) It is commonly used to fine-tune the relevance `_score` for each document by combining the scores from several distinct queries.

Imagine that we want to search for documents((("bool query", "boosting weight of query clauses")))((("weight", "controlling for query clauses"))) about "full-text search," but we want to give more *weight* to documents that also mention "Elasticsearch" or "Lucene." By *more weight*, we mean that documents mentioning "Elasticsearch" or "Lucene" will receive a higher relevance `_score` than those that don't, which means that they will appear higher in the list of results.

A simple `bool` *query* allows us to write this fairly complex logic as follows:

[source,js]

GET /_search { "query": { "bool": { "must": { "match": { "content": { <1> "query": "full text search", "operator": "and" } } }, "should": [<2> { "match": { "content": "Elasticsearch" }}, { "match": { "content": "Lucene" }}] } }

}

// SENSE: 100_Full_Text_Search/25_Boost.json

<1> The `content` field must contain all of the words `full`, `text`, and `search`.

<2> If the `content` field also contains `Elasticsearch` or `Lucene`, the document will receive a higher `_score`.

The more `should` clauses that match, the more relevant the document. So far, so good.

But what if we want to give more weight to the docs that contain `Lucene` and even more weight to the docs containing `Elasticsearch`?

We can control ((("boost parameter")))the relative weight of any query clause by specifying a `boost` value, which defaults to `1`. A `boost` value greater than `1` increases the relative weight of that clause. So we could rewrite the preceding query as follows:

[source,js]

GET /_search { "query": { "bool": { "must": { "match": { <1> "content": { "query": "full text search", "operator": "and" } } }, "should": [{ "match": { "content": { "query": "Elasticsearch", "boost": 3 <2> } }}, { "match": { "content": { "query": "Lucene", "boost": 2 <3> } }}] } }

}

// SENSE: 100_Full_Text_Search/25_Boost.json

<1> These clauses use the default `boost` of `1`.

<2> This clause is the most important, as it has the highest `boost`.

<3> This clause is more important than the default, but not as important as the `Elasticsearch` clause.

[NOTE]

[[boost-normalization]]

The `boost` parameter is used to increase(("boost parameter", "score normalied after boost applied"))) the relative weight of a clause (with a `boost` greater than `1`) or decrease the relative weight (with a `boost` between `0` and `1`), but the increase or decrease is not linear. In other words, a `boost` of `2` does not result in double the `_score`.

Instead, the new `_score` is *normalized* after(("normalization", "score normalied after boost applied"))) the boost is applied. Each type of query has its own normalization algorithm, and the details are beyond the scope of this book. Suffice to say that a higher `boost` value results in a higher `_score`.

If you are implementing your own scoring model not based on TF/IDF and you need more control over the boosting process, you can use the <> to(("function_score query"))) manipulate a document's

boost without the normalization step.

We present other ways of combining queries in the next chapter, <>. But first, let's take a look at the other important feature of queries: text analysis.

=== Controlling Analysis

Queries can find only terms that actually ((("full text search", "controlling analysis")))((("analysis", "controlling")))exist in the inverted index, so it is important to ensure that the same analysis process is applied both to the document at index time, and to the query string at search time so that the terms in the query match the terms in the inverted index.

Although we say *document*, analyzers are determined per field.((("analyzers", "determined per-field"))) Each field can have a different analyzer, either by configuring a specific analyzer for that field or by falling back on the type, index, or node defaults. At index time, a field's value is analyzed by using the configured or default analyzer for that field.

For instance, let's add a new field to `my_index`:

[source,js]

PUT /my_index/_mapping/my_type { "my_type": { "properties": { "english_title": { "type": "string", "analyzer": "english" } } }

}

// SENSE: 100_Full_Text_Search/30_Analysis.json

Now we can compare how values in the `english_title` field and the `title` field are analyzed at index time by using the `analyze` API to analyze the word `Foxes`:

[source,js]

GET /my_index/_analyze?field=my_type.title <1> Foxes

GET /my_index/_analyze?field=my_type.english_title <2>

Foxes

// SENSE: 100_Full_Text_Search/30_Analysis.json

<1> Field `title`, which uses the default `standard` analyzer, will return the term `foxes`.

<2> Field `english_title`, which uses the `english` analyzer, will return the term `fox`.

This means that, were we to run a low-level `term` query for the exact term `fox`, the `english_title` field would match but the `title` field would not.

High-level queries like the `match` query understand field mappings and can apply the correct analyzer for each field being queried.((("match query", "applying appropriate analyzer to each field"))) We can see this in action with ((("validate query API")))the `validate-query` API:

[source,js]

GET /my_index/my_type/_validate/query?explain { "query": { "bool": { "should": [{ "match": { "title": "Foxes"}}, { "match": { "english_title": "Foxes"}}] } }

}

// SENSE: 100_Full_Text_Search/30_Analysis.json

which returns this `explanation`:

```
(title:foxes english_title:fox)
```

The `match` query uses the appropriate analyzer for each field to ensure that it looks for each term in the correct format for that field.

==== Default Analyzers

While we can specify an analyzer at the field level,(((("full text search", "controlling analysis", "default analyzers"))) ((("analyzers", "default"))) how do we determine which analyzer is used for a field if none is specified at the field level?

Analyzers can be specified at several levels. Elasticsearch works through each level until it finds an analyzer that it can use. At index time, the order (((("indexing", "applying analyzers")))is as follows:

- The `analyzer` defined in the field mapping, else
- *The analyzer defined in the `_analyzer` field of the document, else*
- The default `analyzer` for the `type`, which defaults to
- The analyzer named `default` in the index settings, which defaults to
- The analyzer named `default` at node level, which defaults to
- The `standard` analyzer

At search time, the (((("searching", "applying analyzers")))sequence is slightly different:

- *The `analyzer` defined in the query itself, else*
- The `analyzer` defined in the field mapping, else
- The default `analyzer` for the `type`, which defaults to
- The analyzer named `default` in the index settings, which defaults to
- The analyzer named `default` at node level, which defaults to
- The `standard` analyzer

[NOTE]

The two lines in italics in the preceding lists highlight differences in the index time sequence and the search time sequence. The `_analyzer` field allows you to specify a default analyzer for each document (for example, `english`, `french`, `spanish`) while the `analyzer` parameter in the query specifies which analyzer to use on the query string. However, this is not the best way to handle multiple languages

in a single index because of the pitfalls highlighted in <>.

Occasionally, it makes sense to use a different analyzer at index and search time.((("analyzers", "using different analyzers at index and search time"))) For instance, at index time we may want to index synonyms (for example, for every occurrence of `quick`, we also index `fast`, `rapid`, and `speedy`). But at search time, we don't need to search for all of these synonyms. Instead we can just look up the single word that the user has entered, be it `quick`, `fast`, `rapid`, or `speedy`.

To enable this distinction, Elasticsearch also supports ((("index_analyzer parameter")))((("search_analyzer parameter")))the `index_analyzer` and `search_analyzer` parameters, and((("default_search parameter"))) ((("default_index analyzer")))analyzers named `default_index` and `default_search`.

Taking these extra parameters into account, the *full* sequence at index time really looks like this:

- The `index_analyzer` defined in the field mapping, else
- The `analyzer` defined in the field mapping, else
- The analyzer defined in the `_analyzer` field of the document, else

- The default `index_analyzer` for the `type`, which defaults to
- The default `analyzer` for the `type`, which defaults to
- The analyzer named `default_index` in the index settings, which defaults to
- The analyzer named `default` in the index settings, which defaults to
- The analyzer named `default_index` at node level, which defaults to
- The analyzer named `default` at node level, which defaults to
- The `standard` analyzer

And at search time:

- The `analyzer` defined in the query itself, else
- The `search_analyzer` defined in the field mapping, else
- The `analyzer` defined in the field mapping, else
- The default `search_analyzer` for the `type`, which defaults to
- The default `analyzer` for the `type`, which defaults to
- The analyzer named `default_search` in the index settings, which defaults to
- The analyzer named `default` in the index settings, which defaults to
- The analyzer named `default_search` at node level, which defaults to
- The analyzer named `default` at node level, which defaults to
- The `standard` analyzer

==== Configuring Analyzers in Practice

The sheer number of places where you can specify an analyzer is quite overwhelming.((("full text search", "controlling analysis", "configuring analyzers in practice")))((("analyzers", "configuring in practice"))) In practice, though, it is pretty simple.

===== Use index settings, not config files

The first thing to remember is that, even though you may start out using Elasticsearch for a single purpose or a single application such as logging, chances are that you will find more use cases and end up running several distinct applications on the same cluster. Each index needs to be independent and independently configurable. You don't want to set defaults for one use case, only to have to override them for another use case later.

This rules out configuring analyzers at the node level. Additionally, configuring analyzers at the node level requires changing the config file on every node and restarting every node, which becomes a maintenance nightmare. It's a much better idea to keep Elasticsearch running and to manage settings only via the API.

===== Keep it simple

Most of the time, you will know what fields your documents will contain ahead of time. The simplest approach is to set the analyzer for each full-text field when you create your index or add type mappings. While this approach is slightly more verbose, it enables you to easily see which analyzer is being applied to each field.

Typically, most of your string fields will be exact-value `not_analyzed` fields such as tags or enums, plus a handful of full-text fields that will use some default analyzer like `standard` or `english` or some other language. Then you may have one or two fields that need custom analysis: perhaps the `title` field needs to be indexed in a way that supports *find-as-you-type*.

You can set the `default` analyzer in the index to the analyzer you want to use for almost all full-text fields, and just configure the specialized analyzer on the one or two fields that need it. If, in your model, you need a different default analyzer per type, then use the type level `analyzer` setting instead.

[NOTE]

A common work flow for time based data like logging is to create a new index per day on the fly by just indexing into it. While this work flow prevents you from creating your index up front, you can still use http://bit.ly/1ygczeq[index templates]

to specify the settings and mappings that a new index should have.

[[relevance-is-broken]] === Relevance Is Broken!

Before we move on to discussing more-complex queries in <>, let's make a quick detour to explain why we <> with just one primary shard.

Every now and again a new user opens an issue claiming that sorting by relevance(((("relevance", "differences in IDF producing incorrect results"))) is broken and offering a short reproduction: the user indexes a few documents, runs a simple query, and finds apparently less-relevant results appearing above more-relevant results.

To understand why this happens, let's imagine that we create an index with two primary shards and we index ten documents, six of which contain the word `foo`. It may happen that shard 1 contains three of the `foo` documents and shard 2 contains the other three. In other words, our documents are well distributed.

In <>, we described the default similarity algorithm used in Elasticsearch, ((("Term Frequency/Inverse Document Frequency (TF/IDF) similarity algorithm")))called *term frequency / inverse document frequency* or TF/IDF. Term frequency counts the number of times a term appears within the field we are querying in the current document. The more times it appears, the more relevant is this document. The *inverse document frequency* takes((("inverse document frequency")))((("IDF", see="inverse document frequency"))) into account how often a term appears as a percentage of *all the documents in the index*. The more frequently the term appears, the less weight it has.

However, for performance reasons, Elasticsearch doesn't calculate the IDF across all documents in the index.((("shards", "local inverse document frequency (IDF)"))) Instead, each shard calculates a local IDF for the documents contained *in that shard*.

Because our documents are well distributed, the IDF for both shards will be the same. Now imagine instead that five of the `foo` documents are on shard 1, and the sixth document is on shard 2. In this scenario, the term `foo` is very common on one shard (and so of little importance), but rare on the other shard (and so much more important). These differences in IDF can produce incorrect results.

In practice, this is not a problem. The differences between local and global IDF diminish the more documents that you add to the index. With real-world volumes of data, the local IDFs soon even out. The problem is not that relevance is broken but that there is too little data.

For testing purposes, there are two ways we can work around this issue. The first is to create an index with one primary shard, as we did in the section introducing the <>. If you have only one shard, then the local IDF *is* the global IDF.

The second workaround is to add `?search_type=dfs_query_then_fetch` to your search requests. The `dfs` stands(((search*type", "dfs_query_then_fetch")))((("dfs_query_then_fetch search type")))((("DFS (Distributed Frequency Search)"))) for _Distributed Frequency Search*, and it tells Elasticsearch to first retrieve the local IDF from each shard in order to calculate the global IDF across the whole index.

TIP: Don't use `dfs_query_then_fetch` in production. It really isn't required. Just having enough data will ensure that your term frequencies are well distributed. There is no reason to add this extra DFS step to every query that you run.

[[multi-field-search]] == Multifield Search

Queries are seldom simple one-clause `match` queries. ((("multifield search"))) We frequently need to search for the same or different query strings in one or more fields, which means that we need to be able to combine multiple query clauses and their relevance scores in a way that makes sense.

Perhaps we're looking for a book called *War and Peace* by an author called Leo Tolstoy. Perhaps we're searching the Elasticsearch documentation for ``minimum should match,'' which might be in the title or the body of a page. Or perhaps we're searching for users with first name John and last name Smith.

In this chapter, we present the available tools for constructing multiclause searches and how to figure out which solution you should apply to your particular use case.

[[multi-query-strings]] === Multiple Query Strings

The simplest multifield query to deal with is the ((("multifield search", "multiple query strings")))one where we can *map search terms to specific fields*. If we know that *War and Peace* is the title, and Leo Tolstoy is the author, it is easy to write each of these conditions as a `match` clause ((("match clause, mapping search terms to specific fields")))(("bool query", "mapping search terms to specific fields in match clause")))and to combine them with a <>:

[source,js]

GET /_search { "query": { "bool": { "should": [{ "match": { "title": "War and Peace" }}, { "match": { "author": "Leo Tolstoy" }}] } }

}

// SENSE: 110_Multi_Field_Search/05_Multiple_query_strings.json

The `bool` query takes a *more-matches-is-better* approach, so the score from each `match` clause will be added together to provide the final `_score` for each document. Documents that match both clauses will score higher than documents that match just one clause.

Of course, you're not restricted to using just `match` clauses: the `bool` query can wrap any other query type, ((("bool query", "nested bool query in")))including other `bool` queries. We could add a clause to specify that we prefer to see versions of the book that have been translated by specific translators:

[source,js]

GET /_search { "query": { "bool": { "should": [{ "match": { "title": "War and Peace" }}, { "match": { "author": "Leo Tolstoy" }}, { "bool": { "should": [{ "match": { "translator": "Constance Garnett" }}, { "match": { "translator": "Louise Maude" }}] }}] } }

}

// SENSE: 110_Multi_Field_Search/05_Multiple_query_strings.json

Why did we put the translator clauses inside a separate `bool` query? All four `match` queries are `should` clauses, so why didn't we just put the translator clauses at the same level as the title and author clauses?

The answer lies in how the score is calculated.((("relevance scores", "calculation in bool queries"))) The `bool` query runs each `match` query, adds their scores together, then multiplies by the number of matching clauses, and divides by the total number of clauses. Each clause at the same level has the same weight. In the preceding query, the `bool` query containing the translator clauses counts for one-third of the total score. If we had put the translator clauses at the same level as title and author, they would have reduced the contribution of the title and author clauses to one-quarter each.

[[prioritising-clauses]] ==== Prioritizing Clauses

It is likely that an even one-third split between clauses is not what we need for the preceding query. ((("multifield search", "multiple query strings", "prioritizing query clauses")))(("bool query", "prioritizing clauses"))) Probably we're more interested in the title and author clauses then we are in the translator clauses. We need to tune the query to make the title and author clauses relatively more important.

The simplest weapon in our tuning arsenal is the `boost` parameter. To increase the weight of the `title` and `author` fields, give ((("boost parameter", "using to prioritize query clauses")))(("weight", "using boost parameter to prioritize query clauses")))them a `boost` value higher than `1`:

[source,js]

GET /_search { "query": { "bool": { "should": [{ "match": { <1> "title": { "query": "War and Peace", "boost": 2 }}}, { "match": { <1> "author": { "query": "Leo Tolstoy", "boost": 2 }}}, { "bool": { <2> "should": [{ "match": { "translator": "Constance Garnett" }}, { "match": { "translator": "Louise Maude" }}] }}] } }

}

// SENSE: 110_Multi_Field_Search/05_Multiple_query_strings.json

<1> The `title` and `author` clauses have a `boost` value of `2`.

<2> The nested `bool` clause has the default `boost` of `1`.

The `` `best'' value for the `` boost `parameter is most easily determined by trial and error: set a` boost `value, run test queries, repeat. A reasonable range for` boost `lies between` 1 `and` 10 `, maybe` 15`. Boosts higher than that have little more impact because scores are <>.

=== Single Query String

The `bool` query is the mainstay of multiclause queries.((("multifield search", "single query string"))) It works well for many cases, especially when you are able to map different query strings to individual fields.

The problem is that, these days, users expect to be able to type all of their search terms into a single field, and expect that the application will figure out how to give them the right results. It is ironic that the multifield search form is known as *Advanced Search*—it may appear advanced to the user, but it is much simpler to implement.

There is no simple *one-size-fits-all* approach to multiword, multifield queries. To get the best results, you have to *know your data* and know how to use the appropriate tools.

[[know-your-data]] ==== Know Your Data

When your only user input is a single query string, you will encounter three scenarios frequently:

Best fields::

When searching for words that represent a concept, such as `` `brown fox,'' the words mean more together than they do individually. Fields like the `` title `and` body`, while related, can be considered to be in competition with each other. Documents should have as many words as possible in *the same field*, and the score should come from the *best-matching field*.

Most fields::

+

A common technique for fine-tuning relevance is to index the same data into multiple fields, each with its own analysis chain.

The main field may contain words in their stemmed form, synonyms, and words stripped of their *diacritics*, or accents. It is used to match as many documents as possible.

The same text could then be indexed in other fields to provide more-precise matching. One field may contain the unstemmed version, another the original word with accents, and a third might use *shingles* to provide information about <>.

These other fields act as *signals* to increase the relevance score of each

matching document. The *more fields that match*, the better.

Cross fields::

+

For some entities, the identifying information is spread across multiple fields, each of which contains just a part of the whole:

- Person: `first_name` and `last_name`
- Book: `title`, `author`, and `description`
- Address: `street`, `city`, `country`, and `postcode`

In this case, we want to find as many words as possible in *any* of the listed fields. We need to search across multiple fields as if they were one big

field.

All of these are multiword, multifield queries, but each requires a different strategy. We will examine each strategy in turn in the rest of this chapter.

=== Best Fields

Imagine that we have a website that allows (((("multifield search", "best fields queries")))(((("best fields queries")))users to search blog posts, such as these two documents:

[source,js]

PUT /my_index/my_type/1 { "title": "Quick brown rabbits", "body": "Brown rabbits are commonly seen." }

PUT /my_index/my_type/2 { "title": "Keeping pets healthy", "body": "My quick brown fox eats rabbits on a regular basis."

}

// SENSE: 110_Multi_Field_Search/15_Best_fields.json

The user types in the words `` `Brown fox'' and clicks Search. We don't know ahead of time if the user's search terms will be found in the `` title `` or the `` body` field of the post, but it is likely that the user is searching for related words. To our eyes, document 2 appears to be the better match, as it contains both words that we are looking for.

Now we run the following `bool` query:

[source,js]

{ "query": { "bool": { "should": [{ "match": { "title": "Brown fox" }}, { "match": { "body": "Brown fox" }}] } }

}

// SENSE: 110_Multi_Field_Search/15_Best_fields.json

And we find that this query gives document 1 the higher score:

[source,js]

{ "hits": [{ "_id": "1", "_score": 0.14809652, "_source": { "title": "Quick brown rabbits", "body": "Brown rabbits are commonly seen." } }, { "_id": "2", "_score": 0.09256032, "_source": { "title": "Keeping pets healthy", "body": "My quick brown fox eats rabbits on a regular basis." } }]

}

To understand why, think about how the `bool` query (((("bool query", "relevance score calculation")))((("relevance scores", "calculation in bool queries")))calculates its score:

1. It runs both of the queries in the `should` clause.
2. It adds their scores together.
3. It multiplies the total by the number of matching clauses.
4. It divides the result by the total number of clauses (two).

Document 1 contains the word `brown` in both fields, so both `match` clauses are successful and have a score. Document 2 contains both `brown` and `fox` in the `body` field but neither word in the `title` field. The high score from the `body` query is added to the zero score from the `title` query, and multiplied by one-half, resulting in a lower overall score than for document 1.

In this example, the `title` and `body` fields are competing with each other. We want to find the single *best-matching* field.

What if, instead of combining the scores from each field, we used the score from the *best-matching* field as the overall score for the query? This would give preference to a single field that contains *both* of the words we are looking for, rather than the same word repeated in different fields.

[[dis-max-query]] ==== dis_max Query

Instead of the `bool` query, we can use the `dis_max` or *Disjunction Max Query*. Disjunction means *or*((("dis*max (disjunction max) query"))) *(while conjunction means _and*) so the Disjunction Max Query simply means *return documents that match any of these queries, and return the score of the best matching query*:

[source,js]

{ "query": { "dis_max": { "queries": [{ "match": { "title": "Brown fox" }}, { "match": { "body": "Brown fox" }}] } }

}

// SENSE: 110_Multi_Field_Search/15_Best_fields.json

This produces the results that we want:

[source,js]

{ "hits": [{ "_id": "2", "_score": 0.21509302, "_source": { "title": "Keeping pets healthy", "body": "My quick brown fox eats rabbits on a regular basis." } }, { "_id": "1", "_score": 0.12713557, "_source": { "title": "Quick brown rabbits", "body": "Brown rabbits are commonly seen." } }]

}

=== Tuning Best Fields Queries

What would happen if the user((("multifield search", "best fields queries", "tuning")))((("best fields queries", "tuning"))) had searched instead for `quick pets''? Both documents contain the word quick, but only document 2 contains the word pets`. Neither document contains *both words* in the *same field*.

A simple `dis_max` query like the following would (((("dis_max (disjunction max) query")))((("relevance scores", "calculation in dis_max queries")))choose the single best matching field, and ignore the other:

[source,js]

{ "query": { "dis_max": { "queries": [{ "match": { "title": "Quick pets" }}, { "match": { "body": "Quick pets" }}] } }

}

// SENSE: 110_Multi_Field_Search/15_Best_fields.json

[source,js]

{ "hits": [{ "_id": "1", "_score": 0.12713557, <1> "_source": { "title": "Quick brown rabbits", "body": "Brown rabbits are commonly seen." } }, { "_id": "2", "_score": 0.12713557, <1> "_source": { "title": "Keeping pets healthy", "body": "My quick brown fox eats rabbits on a regular basis." } }]

}

<1> Note that the scores are exactly the same.

We would probably expect documents that match on both the `title` field and the `body` field to rank higher than documents that match on just one field, but this isn't the case. Remember: the `dis_max` query simply uses the `_score` from the *single* best-matching clause.

==== tie_breaker

It is possible, however, to((("dis_max (disjunction max) query", "using tie_breaker parameter")))((("relevance scores", "calculation in dis_max queries", "using tie_breaker parameter"))) also take the `_score` from the other matching clauses into account, by specifying (((“tie_breaker parameter")))the `tie_breaker` parameter:

[source,js]

{ "query": { "dis_max": { "queries": [{ "match": { "title": "Quick pets" }}, { "match": { "body": "Quick pets" }}], "tie_breaker": 0.3 } }

}

// SENSE: 110_Multi_Field_Search/15_Best_fields.json

This gives us the following results:

[source,js]

{ "hits": [{ "_id": "2", "_score": 0.14757764, <1> "_source": { "title": "Keeping pets healthy", "body": "My quick brown fox eats

rabbits on a regular basis." } }, { "_id": "1", "_score": 0.124275915, <1> "_source": { "title": "Quick brown rabbits", "body": "Brown rabbits are commonly seen." } }]

}

<1> Document 2 now has a small lead over document 1.

The `tie_breaker` parameter makes the `dis_max` query behave more like a halfway house between `dis_max` and `bool`. It changes the score calculation as follows:

1. Take the `_score` of the best-matching clause.
2. Multiply the score of each of the other matching clauses by the `tie_breaker`.
3. Add them all together and normalize.

With the `tie_breaker`, all matching clauses count, but the best-matching clause counts most.

[NOTE]

The `tie_breaker` can be a floating-point value between `0` and `1`, where `0` uses just the best-matching clause(("tie_breaker parameter", "value of"))) and `1` counts all matching clauses equally. The exact value can be tuned based on your data and queries, but a reasonable value should be close to zero, (for example, `0.1 - 0.4`), in order not to

overwhelm the best-matching nature of `dis_max`.

[[multi-match-query]] === multi_match Query

The `multi_match` query provides (((("multifield search", "multi_match query")))((("multi_match queries")))(("match query", "multi_match queries"))) a convenient shorthand way of running the same query against multiple fields.

[NOTE]

There are several types of `multi_match` query, three of which just happen to coincide with the three scenarios that we listed in

<>: `best_fields`, `most_fields`, and `cross_fields`.

By default, this query runs as type `best_fields`, which means(((("best fields queries", "multi-match queries")))((("dis_max (disjunction max) query", "multi_match query wrapped in"))) that it generates a `match` query for each field and wraps them in a `dis_max` query. This `dis_max` query

[source,js]

{ "dis_max": { "queries": [{ "match": { "title": { "query": "Quick brown fox", "minimum_should_match": "30%" } } }, { "match": { "body": { "query": "Quick brown fox", "minimum_should_match": "30%" } } },], "tie_breaker": 0.3 }

}

could be rewritten more concisely with `multi_match` as follows:

[source,js]

{ "multi_match": { "query": "Quick brown fox", "type": "best_fields", <1> "fields": ["title", "body"], "tie_breaker": 0.3, "minimum_should_match": "30%" <2> }

}

// SENSE: 110_Multi_Field_Search/25_Best_fields.json

<1> The `best_fields` type is the default and can be left out.

<2> Parameters like `minimum_should_match` or `operator` are passed through to the generated `match` queries.

==== Using Wildcards in Field Names

Field names can be specified with wildcards: any field that matches the wildcard pattern((("multi_match queries", "wildcards in field names")))((("wildcards in field names")))((("fields", "wildcards in field names"))) will be included in the search. You could match on the `book_title`, `chapter_title`, and `section_title` fields, with the following:

[source,js]

{ "multi_match": { "query": "Quick brown fox", "fields": "*_title" }

}

==== Boosting Individual Fields

Individual fields can be boosted by using the caret (`^`) syntax: just add `^boost` after the field(("multi_match queries", "boosting individual fields")))((("boost parameter", "boosting individual fields in multi_match queries"))) name, where `boost` is a floating-point number:

[source,js]

{ "multi_match": { "query": "Quick brown fox", "fields": ["*_title", "chapter_title^2"] <1> }

}

<1> The `chapter_title` field has a `boost` of `2`, while the `book_title` and `section_title` fields have a default boost of `1`.

[[most-fields]] === Most Fields

Full-text search is a battle between *recall*—returning all the documents that are (((("most fields queries")))((("multifield search", "most fields queries")))relevant--and *precision*—not returning irrelevant documents. The goal is to present the user with the most relevant documents on the first page of results.

To improve recall, we cast(((("recall", "improving in full text searches"))) the net wide--we include not only documents that match the user's search terms exactly, but also documents that we believe to be pertinent to the query. If a user searches for `` `quick brown fox,'' a document that contains `` fast foxes`` may well be a reasonable result to return.

If the only pertinent document that we have is the one containing `fast foxes`, it will appear at the top of the results list. But of course, if we have 100 documents that contain the words `quick brown fox`, then the `fast foxes` document may be considered less relevant, and we would want to push it further down the list. After including many potential matches, we need to ensure that the best ones rise to the top.

A common technique for fine-tuning full-text relevance((("relevance", "fine-tuning full text relevance"))) is to index the same text in multiple ways, each of which provides a different relevance *signal*. The main field would contain terms in their broadest-matching form to match as many documents as possible. For instance, we could do the following:

- Use a stemmer to index `jumps`, `jumping`, and `jumped` as their root form: `jump`. Then it doesn't matter if the user searches for `jumped`; we could still match documents containing `jumping`.
- Include synonyms like `jump`, `leap`, and `hop`.
- Remove diacritics, or accents: for example, `ésta`, `está`, and `esta` would all be indexed without accents as `esta`.

However, if we have two documents, one of which contains `jumped` and the other `jumping`, the user would probably expect the first document to rank higher, as it contains exactly what was typed in.

We can achieve this by indexing the same text in other fields to provide more-precise matching. One field may contain the unstemmed version, another the original word with diacritics, and a third might use *shingles* to provide information about <>. These other fields act as *signals* that increase the relevance score of each matching document. The more fields that match, the better.

A document is included in the results list if it matches the broad-matching main field. If it also matches the *signal* fields, it gets extra points and is pushed up the results list.

We discuss synonyms, word proximity, partial-matching and other potential signals later in the book, but we will use the simple example of stemmed and unstemmed fields to illustrate this technique.

==== Multifield Mapping

The first thing to do is to set up our ((("most fields queries", "multifield mapping")))((("mapping (types)", "multifield mapping")))field to be indexed twice: once in a stemmed form and once in an unstemmed form. To do this, we will use *multifields*, which we introduced in <>:

[source,js]

DELETE /my_index

PUT /my_index { "settings": { "number_of_shards": 1 }, <1> "mappings": { "my_type": { "properties": { "title": { <2> "type": "string", "analyzer": "english", "fields": { "std": { <3> "type": "string", "analyzer": "standard" } } } } } }

}

// SENSE: 110_Multi_Field_Search/30_Most_fields.json

<1> See <>.

<2> The `title` field is stemmed by the `english` analyzer.

<3> The `title.std` field uses the `standard` analyzer and so is not stemmed.

Next we index some documents:

[source,js]

PUT /my_index/my_type/1 { "title": "My rabbit jumps" }

PUT /my_index/my_type/2

{ "title": "Jumping jack rabbits" }

// SENSE: 110_Multi_Field_Search/30_Most_fields.json

Here is a simple `match` query on the `title` field for `jumping rabbits`:

[source,js]

GET /my_index/_search { "query": { "match": { "title": "jumping rabbits" } }

}

// SENSE: 110_Multi_Field_Search/30_Most_fields.json

This becomes a query for the two stemmed terms `jump` and `rabbit`, thanks to the `english` analyzer. The `title` field of both documents contains both of those terms, so both documents receive the same score:

[source,js]

{ "hits": [{ "_id": "1", "_score": 0.42039964, "_source": { "title": "My rabbit jumps" } }, { "_id": "2", "_score": 0.42039964, "_source": { "title": "Jumping jack rabbits" } }]

}

If we were to query just the `title.std` field, then only document 2 would match. However, if we were to query both fields and to *combine* their scores by using the `bool` query, then both documents would match (thanks to the `title` field) and document 2 would score higher (thanks to the `title.std` field):

[source,js]

GET /my_index/_search { "query": { "multi_match": { "query": "jumping rabbits", "type": "most_fields", <1> "fields": ["title", "title.std"] } }

}

// SENSE: 110_Multi_Field_Search/30_Most_fields.json

<1> We want to combine the scores from all matching fields, so we use the `most_fields` type. This causes the `multi_match` query to wrap the two field-clauses in a `bool` query instead of a `dis_max` query.

[source,js]

{ "hits": [{ "_id": "2", "_score": 0.8226396, <1> "_source": { "title": "Jumping jack rabbits" } }, { "_id": "1", "_score": 0.10741998, <1> "_source": { "title": "My rabbit jumps" } }]

}

<1> Document 2 now scores much higher than document 1.

We are using the broad-matching `title` field to include as many documents as possible--to increase recall--but we use the `title.std` field as a *signal* to push the most relevant results to the top.

The contribution of each field to the final score can be controlled by specifying custom `boost` values. For instance, we could boost the `title` field to make it the most important field, thus reducing the effect of any other signal fields:

[source,js]

GET /my_index/_search { "query": { "multi_match": { "query": "jumping rabbits", "type": "most_fields", "fields": ["title^10", "title.std"] <1> } }

}

// SENSE: 110_Multi_Field_Search/30_Most_fields.json

<1> The `boost` value of `10` on the `title` field makes that field relatively much more important than the `title.std` field.

=== Cross-fields Entity Search

Now we come to a common pattern: cross-fields entity search. (((("cross-fields entity search")))((("multifield search", "cross-fields entity search"))) With entities like `person`, `product`, or `address`, the identifying information is spread across several fields. We may have a `person` indexed as follows:

[source,js]

{ "firstname": "Peter", "lastname": "Smith"

}

Or an address like this:

[source,js]

{ "street": "5 Poland Street", "city": "London", "country": "United Kingdom", "postcode": "W1V 3DG"

}

This sounds a lot like the example we described in <>, but there is a big difference between these two scenarios. In <>, we used a separate query string for each field. In this scenario, we want to search across multiple fields with a *single* query string.

Our user might search for the person `Peter Smith'' or for the address` Poland Street W1V." Each of those words appears in a different field, so using a `dis_max` / `best_fields` query to find the *single* best-matching field is clearly the wrong approach.

==== A Naive Approach

Really, we want to query each field in turn and add up the scores of every field that matches, which sounds like a job for the `bool` query:

[source,js]

{ "query": { "bool": { "should": [{ "match": { "street": "Poland Street W1V" }}, { "match": { "city": "Poland Street W1V" }}, { "match": { "country": "Poland Street W1V" }}, { "match": { "postcode": "Poland Street W1V" }}] } }

}

Repeating the query string for every field soon becomes tedious. We can use the `multi_match` query instead, ((("most fields queries", "problems for entity search")))((("multi_match queries", "most_fields type")))and set the `type` to `most_fields` to tell it to combine the scores of all matching fields:

[source,js]

{ "query": { "multi_match": { "query": "Poland Street W1V", "type": "most_fields", "fields": ["street", "city", "country", "postcode"] } }

}

==== Problems with the most_fields Approach

The `most_fields` approach to entity search has some problems that are not immediately obvious:

- It is designed to find the most fields matching *any* words, rather than to find the most matching words *across all fields*.
- It can't use the `operator` or `minimum_should_match` parameters to reduce the long tail of less-relevant results.
- Term frequencies are different in each field and could interfere with each other to produce badly ordered results.

[[field-centric]] === Field-Centric Queries

All three of the preceding problems stem from ((("field-centric queries")))((("multifield search", "field-centric queries, problems with")))((("most fields queries", "problems with field-centric queries"))) `most_fields` being *field-centric* rather than *term-centric*: it looks for the most matching *fields*, when really what we're interested is the most matching *terms*.

NOTE: The `best_fields` type is also field-centric(((“best fields queries", "problems with field-centric queries"))) and suffers from similar problems.

First we'll look at why these problems exist, and then how we can combat them.

==== Problem 1: Matching the Same Word in Multiple Fields

Think about how the `most_fields` query is executed: Elasticsearch generates a separate `match` query for each field and then wraps these match queries in an outer `bool` query.

We can see this by passing our query through the `validate-query` API:

[source,js]

GET /_validate/query?explain { "query": { "multi_match": { "query": "Poland Street W1V", "type": "most_fields", "fields": ["street", "city", "country", "postcode"] } }

}

// SENSE: 110_Multi_Field_Search/40_Entity_search_problems.json

which yields this `explanation`:

```
(street:poland   street:street   street:w1v)
(city:poland     city:street     city:w1v)
(country:poland  country:street  country:w1v)
(postcode:poland postcode:street postcode:w1v)
```

You can see that a document matching just the word `poland` in *two* fields could score higher than a document matching `poland` and `street` in one field.

==== Problem 2: Trimming the Long Tail

In <>, we talked about(((“and operator", "most fields and best fields queries and")))((("minimum_should_match parameter", "most fields and best fields queries"))) using the `and` operator or the `minimum_should_match` parameter to trim the long tail of almost irrelevant results. Perhaps we could try this:

[source,js]

{ "query": { "multi_match": { "query": "Poland Street W1V", "type": "most_fields", "operator": "and", <1> "fields": ["street", "city", "country", "postcode"] } }

}

// SENSE: 110_Multi_Field_Search/40_Entity_search_problems.json

<1> All terms must be present.

However, with `best_fields` or `most_fields`, these parameters are passed down to the generated `match` queries. The `explanation` for this query shows the following:

```
(+street:poland   +street:street   +street:w1v)
(+city:poland     +city:street     +city:w1v)
(+country:poland  +country:street  +country:w1v)
(+postcode:poland +postcode:street +postcode:w1v)
```

In other words, using the `and` operator means that all words must exist *in the same field*, which is clearly wrong! It is unlikely that any documents would match this query.

==== Problem 3: Term Frequencies

In <>, we explained that the default similarity algorithm used to calculate the relevance score (("term frequency", "problems with field-centric queries")))for each term is TF/IDF:

Term frequency::

```
The more often a term appears in a field in a single document, the more
relevant the document.
```

Inverse document frequency::

```
The more often a term appears in a field in all documents in the index,
the less relevant is that term.
```

When searching against multiple fields, TF/IDF can(("Term Frequency/Inverse Document Frequency (TF/IDF) similarity algorithm", "surprising results when searching against multiple fields"))) introduce some surprising results.

Consider our example of searching for `` `Peter Smith'' using the `` first_name `and` last_name` fields.(("inverse document frequency", "field-centric queries and"))) Peter is a common first name and Smith is a common last name--both will have low IDFs. But what if we have another person in the index whose name is Smith Williams? Smith as a first name is very uncommon and so will have a high IDF!

A simple query like the following may well return Smith Williams above Peter Smith in spite of the fact that the second person is a better match than the first.

[source,js]

{ "query": { "multi_match": { "query": "Peter Smith", "type": "most_fields", "fields": ["*_name"] } }

}

// SENSE: 110_Multi_Field_Search/40_Bad_frequencies.json

The high IDF of `smith` in the first name field can overwhelm the two low IDFs of `peter` as a first name and `smith` as a last name.

==== Solution

These problems only exist because we are dealing with multiple fields. If we were to combine all of these fields into a single field, the problems would vanish. We could achieve this by adding a `full_name` field to our `person` document:

[source,js]

{ "first_name": "Peter", "last_name": "Smith", "full_name": "Peter Smith"

}

When querying just the `full_name` field:

- Documents with more matching words would trump documents with the same word repeated.
- The `minimum_should_match` and `operator` parameters would function as expected.
- The inverse document frequencies for first and last names would be combined so it wouldn't matter whether Smith were a first or last name anymore.

While this would work, we don't like having to store redundant data. Instead, Elasticsearch offers us two solutions--one at index time and one at search time--which we discuss next.

[[custom-all]] === Custom _all Fields

In <>, we explained that the special `_all` field indexes the values from all other fields as one big string.((("_all field", sortas="all field")))((("multifield search", "custom _all fields"))) Having all fields indexed into one field is not terribly flexible, though. It would be nice to have one custom `_all` field for the person's name, and another custom `_all` field for the address.

Elasticsearch provides us with this functionality via the `copy_to` parameter in a field (((("copy_to parameter")))((("mapping (types)", "copy_to parameter")))mapping:

[source,js]

PUT /my_index { "mappings": { "person": { "properties": { "first_name": { "type": "string", "copy_to": "full_name" <1> }, "last_name": { "type": "string", "copy_to": "full_name" <1> }, "full_name": { "type": "string" } } } }

}

// SENSE: 110_Multi_Field_Search/45_Custom_all.json

<1> The values in the `first_name` and `last_name` fields are also copied to the `full_name` field.

With this mapping in place, we can query the `first_name` field for first names, the `last_name` field for last name, or the `full_name` field for first and last names.

NOTE: Mappings of the `first_name` and `last_name` fields have no bearing on how the `full_name` field is indexed. The `full_name` field copies the string values from the other two fields, then indexes them according to the mapping of the `full_name` field only.

=== cross-fields Queries

The custom `_all` approach is a good solution, as long as you thought about setting it up before you indexed your(("multifield search", "cross-fields queries")))(("cross-fields queries"))) documents. However, Elasticsearch also provides a search-time solution to the problem: the `multi_match` query with type `cross_fields`.(("multi*match queries", "cross_fields type"))) The `cross_fields` type takes a term-centric approach, quite different from the field-centric approach taken by `best_fields` and `most_fields`. It treats all of the fields as one big field, and looks for _each term* in *any field*.

To illustrate the difference between field-centric and term-centric queries, look at (("field-centric queries", "differences between term-centric queries and")))(("most fields queries", "explanation for field-centric approach")))the `explanation` for this field-centric `most_fields` query:

[source,js]

GET /_validate/query?explain { "query": { "multi_match": { "query": "peter smith", "type": "most_fields", "operator": "and", <1> "fields": ["first_name", "last_name"] } }

}

// SENSE: 110_Multi_Field_Search/50_Cross_field.json

<1> All terms are required.

For a document to match, both `peter` and `smith` must appear in the same field, either the `first_name` field or the `last_name` field:

```
(+first_name:peter +first_name:smith)
(+last_name:peter  +last_name:smith)
```

A *term-centric* approach would use this logic instead:

```
+(first_name:peter last_name:peter)
+(first_name:smith last_name:smith)
```

In other words, the term `peter` must appear in either field, and the term `smith` must appear in either field.

The `cross_fields` type first analyzes the query string to produce a list of terms, and then it searches for each term in any field. That difference alone solves two of the three problems that we listed in <>, leaving us just with the issue of differing inverse document frequencies.

Fortunately, the `cross_fields` type solves this too, as can be seen from this `validate-query` request:

[source,js]

GET /_validate/query?explain { "query": { "multi_match": { "query": "peter smith", "type": "cross_fields", <1> "operator": "and", "fields": ["first_name", "last_name"] } }

}

// SENSE: 110_Multi_Field_Search/50_Cross_field.json

<1> Use `cross_fields` term-centric matching.

It solves the term-frequency problem by *blending* inverse document frequencies across fields: ((("cross-fields queries", "blending inverse document frequencies across fields")))((("inverse document frequency", "blending across fields in cross-fields queries")))

```
+blended("peter", fields: [first_name, last_name])
+blended("smith", fields: [first_name, last_name])
```

In other words, it looks up the IDF of `smith` in both the `first_name` and the `last_name` fields and uses the minimum of the two as the IDF for both fields. The fact that `smith` is a common last name means that it will be treated as a common first name too.

[NOTE]

For the `cross_fields` query type to work optimally, all fields should have the same analyzer.((("analyzers", "in cross-fields queries")))((("cross-fields queries", "analyzers in"))) Fields that share an analyzer are grouped together as blended fields.

If you include fields with a different analysis chain, they will be added to the query in the same way as for `best_fields`. For instance, if we added the `title` field to the preceding query (assuming it uses a different analyzer), the explanation would be as follows:

```
(+title:peter +title:smith)
(
  +blended("peter", fields: [first_name, last_name])
  +blended("smith", fields: [first_name, last_name])
)
```

This is particularly important when using the `minimum_should_match` and

`operator` parameters.

==== Per-Field Boosting

One of the advantages of using the `cross_fields` query over <> is that you ((("cross-fields queries", "per-field boosting")))((("boosting", "per-field boosting in cross-fields queries")))can boost individual fields at query time.

For fields of equal value like `first_name` and `last_name`, this generally isn't required, but if you were searching for books using the `title` and `description` fields, you might want to give more weight to the `title` field. This can be done as described before with the caret (`^`) syntax:

[source,js]

GET /books/_search { "query": { "multi_match": { "query": "peter smith", "type": "cross_fields", "fields": ["title^2", "description"] <1> } }

}

<1> The `title` field has a boost of `2`, while the `description` field has the default boost of `1`.

The advantage of being able to boost individual fields should be weighed against the cost of querying multiple fields instead of querying a single custom `_all` field. Use whichever of the two solutions that delivers the most bang for your buck.

=== Exact-Value Fields

The final topic that we should touch on before leaving multifield queries is that of exact-value `not_analyzed` fields. ((("not_analyzed fields", "exact value, in multi-field queries")))((("multifield search", "exact value fields")))((("exact values", "exact value not_analyzed fields in multifield search")))((("analyzed fields", "avoiding mixing with not analyzed fields in multi_match queries"))) It is not useful to mix `not_analyzed` fields with `analyzed` fields in `multi_match` queries.

The reason for this can be demonstrated easily by looking at a query explanation. Imagine that we have set the `title` field to be `not_analyzed`:

[source,js]

GET /_validate/query?explain { "query": { "multi_match": { "query": "peter smith", "type": "cross_fields", "fields": ["title", "first_name", "last_name"] } }

}

// SENSE: 110_Multi_Field_Search/55_Not_analyzed.json

Because the `title` field is not analyzed, it searches that field for a single term consisting of the whole query string!

```
title:peter smith
(
    blended("peter", fields: [first_name, last_name])
    blended("smith", fields: [first_name, last_name])
)
```

That term clearly does not exist in the inverted index of the `title` field, and can never be found. Avoid using `not_analyzed` fields in `multi_match` queries.

[[proximity-matching]] == Proximity Matching

Standard full-text search with TF/IDF treats documents, or at least each field within a document, as a big *bag of words*. ((("proximity matching"))) The `match` query can tell us whether that bag contains our search terms, but that is only part of the story. It can't tell us anything about the relationship between words.

Consider the difference between these sentences:

- Sue ate the alligator.
- The alligator ate Sue.
- Sue never goes anywhere without her alligator-skin purse.

A `match` query for `sue alligator` would match all three documents, but it doesn't tell us whether the two words form part of the same idea, or even the same paragraph.

Understanding how words relate to each other is a complicated problem, and we can't solve it by just using another type of query, but we can at least find words that appear to be related because they appear near each other or even right next to each other.

Each document may be much longer than the examples we have presented: `Sue` and `alligator` may be separated by paragraphs of other text. Perhaps we still want to return these documents in which the words are widely separated, but we want to give documents in which the words are close together a higher relevance score.

This is the province of *phrase matching*, or *proximity matching*.

[TIP]

In this chapter, we are using the same example documents that we used for the <>.

==========

[[phrase-matching]] === Phrase Matching

In the same way that the `match` query is the go-to query for standard full-text search, the `match_phrase` query(((("proximity matching", "phrase matching")))((("phrase matching"))))((("match_phrase query"))) is the one you should reach for when you want to find words that are near each other:

[source,js]

GET /my_index/my_type/_search { "query": { "match_phrase": { "title": "quick brown fox" } }

}

// SENSE: 120_Proximity_Matching/05_Match_phrase_query.json

Like the `match` query, the `match_phrase` query first analyzes the query string to produce a list of terms. It then searches for all the terms, but keeps only documents that contain *all* of the search terms, in the same *positions* relative to each other. A query for the phrase `quick fox` would not match any of our documents, because no document contains the word `quick` immediately followed by `fox`.

[TIP]

The `match_phrase` query can also be written as a `match` query with type `phrase`:

[source,js]

"match": { "title": { "query": "quick brown fox", "type": "phrase" }

}

// SENSE: 120_Proximity_Matching/05_Match_phrase_query.json

==========

==== Term Positions

When a string is analyzed, the analyzer returns not(((("phrase matching", "term positions")))((("match*phrase query", "position of terms")))((("position-aware matching")))) only a list of terms, but also the _position*, or order, of each term in the original string:

[source,js]

GET /_analyze?analyzer=standard

Quick brown fox

// SENSE: 120_Proximity_Matching/05_Term_positions.json

This returns the following:

[role="pagebreak-before"]

[source,js]

{ "tokens": [{ "token": "quick", "start_offset": 0, "end_offset": 5, "type": "", "position": 1 <1> }, { "token": "brown", "start_offset": 6, "end_offset": 11, "type": "", "position": 2 <1> }, { "token": "fox", "start_offset": 12, "end_offset": 15, "type": "", "position": 3 <1> }]

}

<1> The `position` of each term in the original string.

Positions can be stored in the inverted index, and position-aware queries like the `match_phrase` query can use them to match only documents that contain all the words in exactly the order specified, with no words in-between.

==== What Is a Phrase

For a document to be considered a(((("match_phrase query", "documents matching a phrase")))(((("phrase matching", "criteria for matching documents"))) match for the phrase ``quick brown fox," the following must be true:

- `quick`, `brown`, and `fox` must all appear in the field.
- The position of `brown` must be `1` greater than the position of `quick`.
- The position of `fox` must be `2` greater than the position of `quick`.

If any of these conditions is not met, the document is not considered a match.

[TIP]

Internally, the `match_phrase` query uses the low-level `span` query family to do position-aware matching. ((("match_phrase query", "use of span queries for position-aware matching")))((("span queries")))Span queries are term-level queries, so they have no analysis phase; they search for the exact term specified.

Thankfully, most people never need to use the `span` queries directly, as the `match_phrase` query is usually good enough. However, certain specialized fields, like patent searches, use these low-level queries to perform very specific, carefully constructed positional searches.

==========

[[slop]] === Mixing It Up

Requiring exact-phrase matches ((("proximity matching", "slop parameter")))may be too strict a constraint. Perhaps we *do* want documents that contain `quick brown fox'' to be considered a match for the query` quick fox,'' even though the positions aren't exactly equivalent.

We can introduce a degree ((("slop parameter")))of flexibility into phrase matching by using the `slop` parameter:

[source,js]

GET /my_index/my_type/_search { "query": { "match_phrase": { "title": { "query": "quick fox", "slop": 1 } } }

}

// SENSE: 120_Proximity_Matching/10_Slop.json

The `slop` parameter tells the `match_phrase` query how(((match*phrase query", "slop parameter"))) far apart terms are allowed to be while still considering the document a match. By _how far apart* we mean *how many times do you need to move a term in order to make the query and document match*?

We'll start with a simple example. To make the query `quick fox` match a document containing `quick brown fox` we need a `slop` of just `1`:

```
            Pos 1         Pos 2         Pos 3
-------------------------------------------------
Doc:        quick         brown         fox
-------------------------------------------------
Query:      quick         fox
Slop 1:     quick                 ↳     fox
```

Although all words need to be present in phrase matching, even when using `slop`, the words don't necessarily need to be in the same sequence in order to match. With a high enough `slop` value, words can be arranged in any order.

To make the query `fox quick` match our document, we need a `slop` of `3`:

```
            Pos 1         Pos 2         Pos 3
-------------------------------------------------
Doc:        quick         brown         fox
-------------------------------------------------
Query:      fox           quick
Slop 1:     fox|quick  ↵  <1>
Slop 2:     quick      ↳  fox
Slop 3:     quick                 ↳     fox
```

<1> Note that `fox` and `quick` occupy the same position in this step. Switching word order from `fox quick` to `quick fox` thus requires two steps, or a `slop` of `2`.

=== Multivalue Fields

A curious thing can happen when you try to use phrase matching on multivalue fields. ((("proximity matching", "on multivalue fields")))((("match_phrase query", "on multivalue fields"))) Imagine that you index this document:

[source,js]

PUT /my_index/groups/1 { "names": ["John Abraham", "Lincoln Smith"]

}

// SENSE: 120_Proximity_Matching/15_Multi_value_fields.json

Then run a phrase query for `Abraham Lincoln`:

[source,js]

GET /my_index/groups/_search { "query": { "match_phrase": { "names": "Abraham Lincoln" } }

}

// SENSE: 120_Proximity_Matching/15_Multi_value_fields.json

Surprisingly, our document matches, even though `Abraham` and `Lincoln` belong to two different people in the `names` array. The reason for this comes down to the way arrays are indexed in Elasticsearch.

When `John Abraham` is analyzed, it produces this:

- Position 1: `john`
- Position 2: `abraham`

Then when `Lincoln Smith` is analyzed, it produces this:

- Position 3: `lincoln`
- Position 4: `smith`

In other words, Elasticsearch produces exactly the same list of tokens as it would have for the single string `John Abraham Lincoln Smith`. Our example query looks for `abraham` directly followed by `lincoln`, and these two terms do indeed exist, and they are right next to each other, so the query matches.

Fortunately, there is a simple workaround for cases like these, called the `position_offset_gap`, which((("mapping (types)", "position_offset_gap")))((("position_offset_gap"))) we need to configure in the field mapping:

[source,js]

DELETE /my_index/groups/ <1>

PUT /my_index/_mapping/groups <2> { "properties": { "names": { "type": "string", "position_offset_gap": 100 } }

}

// SENSE: 120_Proximity_Matching/15_Multi_value_fields.json

<1> First delete the `groups` mapping and all documents of that type.

<2> Then create a new `groups` mapping with the correct values.

The `position_offset_gap` setting tells Elasticsearch that it should increase the current term `position` by the specified value for every new array element. So now, when we index the array of names, the terms are emitted with the following positions:

- Position 1: `john`
- Position 2: `abraham`
- Position 103: `lincoln`
- Position 104: `smith`

Our phrase query would no longer match a document like this because `abraham` and `lincoln` are now 100 positions apart. You would have to add a `slop` value of 100 in order for this document to match.

=== Closer Is Better

Whereas a phrase query simply excludes documents that don't contain the exact query phrase, a *proximity query*—a ((("proximity matching", "proximity queries")))(((("slop parameter", "proximity queries and")))phrase query where `slop` is greater than `0` —incorporates the proximity of the query terms into the final relevance `_score`. By setting a high `slop` value like `50` or `100`, you can exclude documents in which the words are really too far apart, but give a higher score to documents in which the words are closer together.

The following proximity query for `quick dog` matches both documents that contain the words `quick` and `dog`, but gives a higher score to the document(((("relevance scores", "for proximity queries"))) in which the words are nearer to each other:

[source,js]

POST /my_index/my_type/_search { "query": { "match_phrase": { "title": { "query": "quick dog", "slop": 50 <1> } } }

}

// SENSE: 120_Proximity_Matching/20_Scoring.json

<1> Note the high `slop` value.

[source,js]

{ "hits": [{ "_id": "3", "_score": 0.75, <1> "_source": { "title": "The quick brown fox jumps over the quick dog" } }, { "_id": "2", "_score": 0.28347334, <2> "_source": { "title": "The quick brown fox jumps over the lazy dog" } }]

}

<1> Higher score because `quick` and `dog` are close together

<2> Lower score because `quick` and `dog` are further apart

[[proximity-relevance]] === Proximity for Relevance

Although proximity queries are useful, the fact that they require all terms to be present can make them overly strict. ((("proximity matching", "using for relevance")))((("relevance", "proximity queries for"))) It's the same issue that we discussed in <> in <>: if six out of seven terms match, a document is probably relevant enough to be worth showing to the user, but the `match_phrase` query would exclude it.

Instead of using proximity matching as an absolute requirement, we can use it as a *signal*—as one of potentially many queries, each of which contributes to the overall score for each document (see <>).

The fact that we want to add together the scores from multiple queries implies that we should combine them by using the `bool` query.((("bool query", "proximity query for relevance in")))

We can use a simple `match` query as a `must` clause. This is the query that will determine which documents are included in our result set. We can trim the long tail with the `minimum_should_match` parameter. Then we can add other, more specific queries as `should` clauses. Every one that matches will increase the relevance of the matching docs.

[source,js]

GET /my_index/my_type/_search { "query": { "bool": { "must": { "match": { <1> "title": { "query": "quick brown fox", "minimum_should_match": "30%" } } }, "should": { "match_phrase": { <2> "title": { "query": "quick brown fox", "slop": 50 } } } } }

}

// SENSE: 120_Proximity_Matching/25_Relevance.json

<1> The `must` clause includes or excludes documents from the result set.

<2> The `should` clause increases the relevance score of those documents that match.

We could, of course, include other queries in the `should` clause, where each query targets a specific aspect of relevance.

[role="pagebreak-before"] === Improving Performance

Phrase and proximity queries are more (((("proximity matching", "improving performance")))((("phrase matching", "improving performance")))expensive than simple `match` queries. Whereas a `match` query just has to look up terms in the inverted index, a `match_phrase` query has to calculate and compare the positions of multiple possibly repeated terms.

The http://people.apache.org/~mikemccand/lucenebench/[Lucene nightly benchmarks] show that a simple `term` query is about 10 times as fast as a phrase query, and about 20 times as fast as a proximity query (a phrase query with `slop`). And of course, this cost is paid at search time instead of at index time.

[TIP]

Usually the extra cost of phrase queries is not as scary as these numbers suggest. Really, the difference in performance is a testimony to just how fast a simple `term` query is. Phrase queries on typical full-text data usually complete within a few milliseconds, and are perfectly usable in practice, even on a busy cluster.

In certain pathological cases, phrase queries can be costly, but this is unusual. An example of a pathological case is DNA sequencing, where there are many many identical terms repeated in many positions. Using higher `slop` values in this case results in a huge growth in the number of position calculations.

==========

So what can we do to limit the performance cost of phrase and proximity queries? One useful approach is to reduce the total number of documents that need to be examined by the phrase query.

[[rescore-api]] ==== Rescoring Results

In <>, we discussed using proximity queries just for relevance purposes, not to include or exclude results from the result set. ((("relevance scores", "rescoring results for top-N documents with proximity query"))) A query may match millions of results, but chances are that our users are interested in only the first few pages of results.

A simple `match` query will already have ranked documents that contain all search terms near the top of the list. Really, we just want to rerank the *top results* to give an extra relevance bump to those documents that also match the phrase query.

The `search` API supports exactly this functionality via *rescoring*.(((("rescoring"))) The rescore phase allows you to apply a more expensive scoring algorithm--like a `phrase` query--to just the top `K` results from each shard. These top results are then resorted according to their new scores.

The request looks like this:

[source,js]

GET /my_index/my_type/_search { "query": { "match": { <1> "title": { "query": "quick brown fox", "minimum_should_match": "30%" } } }, "rescore": { "window_size": 50, <2> "query": { <3> "rescore_query": { "match_phrase": { "title": { "query": "quick brown fox", "slop": 50 } } } } }

}

// SENSE: 120_Proximity_Matching/30_Performance.json

<1> The `match` query decides which results will be included in the final result set and ranks results according to TF/IDF. ((("window_size parameter")))

<2> The `window_size` is the number of top results to rescore, per shard.

<3> The only rescoring algorithm currently supported is another query, but there are plans to add more algorithms later.

[[shingles]] === Finding Associated Words

As useful as phrase and proximity queries can be, they still have a downside. They are overly strict: all terms must be present for a phrase query to match, even when using `slop`.(("proximity matching", "finding associated words", range="startofrange", id="ix_proxmatchassoc")))

The flexibility in word ordering that you gain with `slop` also comes at a price, because you lose the association between word pairs. While you can identify documents in which `sue`, `alligator`, and `ate` occur close together, you can't tell whether *Sue ate* or the *alligator ate*.

When words are used in conjunction with each other, they express an idea that is bigger or more meaningful than each word in isolation. The two clauses *I'm not happy I'm working* and *I'm happy I'm not working* contain the sames words, in close proximity, but have quite different meanings.

If, instead of indexing each word independently, we were to index pairs of words, then we could retain more of the context in which the words were used.

For the sentence `Sue ate the alligator`, we would not only index each word (or *unigram*) as(("unigrams")) a term

```
["sue", "ate", "the", "alligator"]
```

but also each word *and its neighbor* as single terms:

```
["sue ate", "ate the", "the alligator"]
```

These word (("bigrams"))pairs (or *bigrams*) are (("shingles"))known as *shingles*.

[TIP]

Shingles are not restricted to being pairs of words; you could index word triplets (*trigrams*) as (("trigrams"))well:

```
["sue ate the", "ate the alligator"]
```

Trigrams give you a higher degree of precision, but greatly increase the number of unique terms in the index. Bigrams are sufficient for most use cases.

==========

Of course, shingles are useful only if the user enters the query in the same order as in the original document; a query for `sue alligator` would match the individual words but none of our shingles.

Fortunately, users tend to express themselves using constructs similar to those that appear in the data they are searching. But this point is an important one: it is not enough to index just bigrams; we still need unigrams, but we can use matching bigrams as a signal to increase the relevance score.

==== Producing Shingles

Shingles need to be created at index time as part of the analysis process.(("shingles", "producing at index time")) We could index both unigrams and bigrams into a single field, but it is cleaner to keep unigrams and bigrams in separate fields that can be queried independently. The unigram field would form the basis of our search, with the bigram field being used to boost relevance.

First, we need to create an analyzer that uses the `shingle` token filter:

[source,js]

DELETE /my_index

PUT /my_index { "settings": { "number_of_shards": 1, <1> "analysis": { "filter": { "my_shingle_filter": { "type": "shingle", "min_shingle_size": 2, <2> "max_shingle_size": 2, <2> "output_unigrams": false <3> } }, "analyzer": { "my_shingle_analyzer": { "type": "custom", "tokenizer": "standard", "filter": ["lowercase", "my_shingle_filter" <4>] } } } }

}

// SENSE: 120_Proximity_Matching/35_Shingles.json

<1> See <>.

<2> The default min/max shingle size is `2` so we don't really need to set these.

<3> The `shingle` token filter outputs unigrams by default, but we want to keep unigrams and bigrams separate.

<4> The `my_shingle_analyzer` uses our custom `my_shingles_filter` token filter.

First, let's test that our analyzer is working as expected with the `analyze` API:

[source,js]

GET /my_index/_analyze?analyzer=my_shingle_analyzer

Sue ate the alligator

Sure enough, we get back three terms:

- `sue ate`
- `ate the`
- `the alligator`

Now we can proceed to setting up a field to use the new analyzer.

==== Multifields

We said that it is cleaner to index unigrams and bigrams separately, so we will create the `title` field (("multifields")))as a multifield (see <>):

[source,js]

PUT /my_index/_mapping/my_type { "my_type": { "properties": { "title": { "type": "string", "fields": { "shingles": { "type": "string", "analyzer": "my_shingle_analyzer" } } } } }

}

With this mapping, values from our JSON document in the field `title` will be indexed both as unigrams (`title`) and as bigrams (`title.shingles`), meaning that we can query these fields independently.

And finally, we can index our example documents:

[source,js]

POST /my_index/my_type/_bulk { "index": { "_id": 1 }} { "title": "Sue ate the alligator" } { "index": { "_id": 2 }} { "title": "The alligator ate Sue" } { "index": { "_id": 3 }}

{ "title": "Sue never goes anywhere without her alligator skin purse" }

==== Searching for Shingles

To understand the benefit (((("shingles", "searching for")))that the `shingles` field adds, let's first look at the results from a simple `match` query for ``The hungry alligator ate Sue'':

[source,js]

GET /my_index/my_type/_search { "query": { "match": { "title": "the hungry alligator ate sue" } }

}

This query returns all three documents, but note that documents 1 and 2 have the same relevance score because they contain the same words:

[source,js]

{ "hits": [{ "_id": "1", "_score": 0.44273707, <1> "_source": { "title": "Sue ate the alligator" } }, { "_id": "2", "_score": 0.44273707, <1> "_source": { "title": "The alligator ate Sue" } }, { "_id": "3", <2> "_score": 0.046571054, "_source": { "title": "Sue never goes anywhere without her alligator skin purse" } }]

}

<1> Both documents contain `the` , `alligator` , and `ate` and so have the same score.

<2> We could have excluded document 3 by setting the `minimum_should_match` parameter. See <>.

Now let's add the `shingles` field into the query. Remember that we want matches on the `shingles` field to act as a signal--to increase the relevance score--so we still need to include the query on the main `title` field:

[source,js]

GET /my_index/my_type/_search { "query": { "bool": { "must": { "match": { "title": "the hungry alligator ate sue" } }, "should": { "match": { "title.shingles": "the hungry alligator ate sue" } } } }

}

We still match all three documents, but document 2 has now been bumped into first place because it matched the shingled term `ate sue` .

[source,js]

{ "hits": [{ "_id": "2", "_score": 0.4883322, "_source": { "title": "The alligator ate Sue" } }, { "_id": "1", "_score": 0.13422975, "_source": { "title": "Sue ate the alligator" } }, { "_id": "3", "_score": 0.014119488, "_source": { "title": "Sue never goes anywhere without her alligator skin purse" } }]

}

Even though our query included the word `hungry`, which doesn't appear in any of our documents, we still managed to use word proximity to return the most relevant document first.

==== Performance

Not only are shingles more flexible than phrase queries,(("shingles", "better performance than phrase queries"))) but they perform better as well. Instead of paying the price of a phrase query every time you search, queries for shingles are just as efficient as a simple `match` query. A small price is paid at index time, because more terms need to be indexed, which also means that fields with shingles use more disk space. However, most applications write once and read many times, so it makes sense to optimize for fast queries.

This is a theme that you will encounter frequently in Elasticsearch: enables you to achieve a lot at search time, without requiring any up-front setup. Once you understand your requirements more clearly, you can achieve better results with better performance by modeling your data correctly at index time. ((("proximity matching", "finding associated words", range="endofrange", startref ="ix_proxmatchassoc")))

[[partial-matching]] == Partial Matching

A keen observer will notice that all the queries so far in this book have operated on whole terms.((("partial matching"))) To match something, the smallest unit had to be a single term. You can find only terms that exist in the inverted index.

But what happens if you want to match parts of a term but not the whole thing? *Partial matching* allows users to specify a portion of the term they are looking for and find any words that contain that fragment.

The requirement to match on part of a term is less common in the full-text search-engine world than you might think. If you have come from an SQL background, you likely have, at some stage of your career, implemented a *poor man's full-text search* using SQL constructs like this:

[source,js]

```
WHERE text LIKE "*quick*"
  AND text LIKE "*brown*"
  AND text LIKE "*fox*" <1>
```

<1> `*fox*` would match `fox'' and` foxes."

Of course, with Elasticsearch, we have the analysis process and the inverted index that remove the need for such brute-force techniques. To handle the case of matching both `fox'' and` foxes," we could simply use a stemmer to index words in their root form. There is no need to match partial terms.

That said, on some occasions partial matching can be useful. Common use ((("partial matching", "common use cases")))cases include the following:

- Matching postal codes, product serial numbers, or other `not_analyzed` values that start with a particular prefix or match a wildcard pattern or even a regular expression
- *search-as-you-type*—displaying the most likely results before the user has finished typing the search terms
- Matching in languages like German or Dutch, which contain long compound words, like *Weltgesundheitsorganisation* (World Health Organization)

We will start by examining prefix matching on exact-value `not_analyzed` fields.

=== Postcodes and Structured Data

We will use United Kingdom postcodes (postal codes in the United States) to illustrate how((("partial matching", "postcodes and structured data"))) to use partial matching with structured data. UK postcodes have a well-defined structure. For instance, the postcode `W1V 3DG` can((("postcodes (UK), partial matching with"))) be broken down as follows:

- `W1V` : This outer part identifies the postal area and district:

`W` **indicates the area (one or two letters)** `1V` indicates the district (one or two numbers, possibly followed by a letter

- `3DG` : This inner part identifies a street or building:

`3` **indicates the sector (one number)** `DG` indicates the unit (two letters)

Let's assume that we are indexing postcodes as exact-value `not_analyzed` fields, so we could create our index as follows:

[source,js]

PUT /my_index { "mappings": { "address": { "properties": { "postcode": { "type": "string", "index": "not_analyzed" } } } }

}

// SENSE: 130_Partial_Matching/10_Prefix_query.json

And index some ((("indexing", "postcodes")))postcodes:

[source,js]

PUT /my_index/address/1 { "postcode": "W1V 3DG" }

PUT /my_index/address/2 { "postcode": "W2F 8HW" }

PUT /my_index/address/3 { "postcode": "W1F 7HW" }

PUT /my_index/address/4 { "postcode": "WC1N 1LZ" }

PUT /my_index/address/5

{ "postcode": "SW5 0BE" }

// SENSE: 130_Partial_Matching/10_Prefix_query.json

Now our data is ready to be queried.

[[prefix-query]] === prefix Query

To find all postcodes beginning with `W1`, we could use a ((("prefix query")))((("postcodes (UK), partial matching with", "prefix query")))simple `prefix` query:

[source,js]

GET /my_index/address/_search { "query": { "prefix": { "postcode": "W1" } }

}

// SENSE: 130_Partial_Matching/10_Prefix_query.json

The `prefix` query is a low-level query that works at the term level. It doesn't analyze the query string before searching. It assumes that you have passed it the exact prefix that you want to find.

[TIP]

By default, the `prefix` query does no relevance scoring. It just finds matching documents and gives them all a score of `1`. Really, it behaves more like a filter than a query. The only practical difference between the `prefix` query and the `prefix` filter is that the filter can be cached.

==========

Previously, we said that `` `you can find only terms that exist in the inverted index,'' but we haven't done anything special to index these postcodes; each postcode is simply indexed as the exact value specified in each document. So how does the `` prefix` query work?

[role="pagebreak-after"] Remember that the inverted index consists((("inverted index", "for postcodes"))) of a sorted list of unique terms (in this case, postcodes). For each term, it lists the IDs of the documents containing that term in the *postings list*. The inverted index for our example documents looks something like this:

```
Term:          Doc IDs:
-------------------------
"SW5 0BE"    |  5
"W1F 7HW"    |  3
"W1V 3DG"    |  1
"W2F 8HW"    |  2
"WC1N 1LZ"   |  4
-------------------------
```

To support prefix matching on the fly, the query does the following:

1. Skips through the terms list to find the first term beginning with `W1`.
2. Collects the associated document IDs.
3. Moves to the next term.
4. If that term also begins with `W1`, the query repeats from step 2; otherwise, we're finished.

While this works fine for our small example, imagine that our inverted index contains a million postcodes beginning with `W1`. The prefix query would need to visit all one million terms in order to calculate the result!

And the shorter the prefix, the more terms need to be visited. If we were to look for the prefix `W` instead of `W1`, perhaps we would match 10 million terms instead of just one million.

CAUTION: The `prefix` query or filter are useful for ad hoc prefix matching, but should be used with care. ((("prefix query", "caution with"))) They can be used freely on fields with a small number of terms, but they scale poorly and can put your

cluster under a lot of strain. Try to limit their impact on your cluster by using a long prefix; this reduces the number of terms that need to be visited.

Later in this chapter, we present an alternative index-time solution that makes prefix matching much more efficient. But first, we'll take a look at two related queries: the `wildcard` and `regexp` queries.

=== wildcard and regexp Queries

The `wildcard` query is a low-level, term-based query (((("wildcard query")))((("partial matching", "wildcard and regexp queries")))similar in nature to the `prefix` query, but it allows you to specify a pattern instead of just a prefix. It uses the standard shell wildcards: `?` matches any character, and `*` matches zero or more characters.((("postcodes (UK), partial matching with", "wildcard queries")))

This query would match the documents containing `W1F 7HW` and `W2F 8HW`:

[source,js]

GET /my_index/address/_search { "query": { "wildcard": { "postcode": "W?F*HW" <1> } }

}

// SENSE: 130_Partial_Matching/15_Wildcard_regexp.json

<1> The `?` matches the `1` and the `2`, while the `*` matches the space and the `7` and `8`.

Imagine now that you want to match all postcodes just in the `W` area. A prefix match would also include postcodes starting with `WC`, and you would have a similar problem with a wildcard match. We want to match only postcodes that begin with a `W`, followed by a number.((("postcodes (UK), partial matching with", "regexp query")))((("regexp query"))) The `regexp` query allows you to write these more complicated patterns:

[source,js]

GET /my_index/address/_search { "query": { "regexp": { "postcode": "W[0-9].+" <1> } }

}

// SENSE: 130_Partial_Matching/15_Wildcard_regexp.json

<1> The regular expression says that the term must begin with a `W`, followed by any number from 0 to 9, followed by one or more other characters.

The `wildcard` and `regexp` queries work in exactly the same way as the `prefix` query. They also have to scan the list of terms in the inverted index to find all matching terms, and gather document IDs term by term. The only difference between them and the `prefix` query is that they support more-complex patterns.

This means that the same caveats apply. Running these queries on a field with many unique terms can be resource intensive indeed. Avoid using a pattern that starts with a wildcard (for example, `*foo` or, as a regexp, `.*foo`).

Whereas prefix matching can be made more efficient by preparing your data at index time, wildcard and regular expression matching can be done only at query time. These queries have their place but should be used sparingly.

[CAUTION]

The `prefix`, `wildcard`, and `regexp` queries operate on terms. If you use them to query an `analyzed` field, they will examine each term in the field, not the field as a whole.((("prefix query", "on analyzed fields")))((("wildcard query", "on analyzed fields")))((("regexp query", "on analyzed fields")))((("analyzed fields", "prefix, wildcard, and regexp queries on")))

For instance, let's say that our `title` field contains `` `Quick brown fox'' which produces the terms `` quick `,` brown `, and` fox`.

This query would match:

[source,json]

{ "regexp": { "title": "br.*" }}

But neither of these queries would match:

[source,json]

{ "regexp": { "title": "Qu.*" }} <1>

{ "regexp": { "title": "quick br*" }} <2>

<1> The term in the index is `quick`, not `Quick`.

<2> `quick` and `brown` are separate terms.

==========

=== Query-Time Search-as-You-Type

Leaving postcodes behind, let's take a look at how prefix matching can help with full-text queries. (((("partial matching", "query time search-as-you-type"))) Users have become accustomed to seeing search results before they have finished typing their query--so-called *instant search*, or *search-as-you-type*. (((("search-as-you-type")))((("instant search"))) Not only do users receive their search results in less time, but we can guide them toward results that actually exist in our index.

For instance, if a user types in `johnnie walker bl`, we would like to show results for Johnnie Walker Black Label and Johnnie Walker Blue Label before they can finish typing their query.

As always, there are more ways than one to skin a cat! We will start by looking at the way that is simplest to implement. You don't need to prepare your data in any way; you can implement *search-as-you-type* at query time on any full-text field.

In <>, we introduced the `match_phrase` query, which matches all the specified words in the same positions relative to each other. For-query time search-as-you-type, we can use a specialization of this query, called ((("prefix query", "match_phrase_prefix query")))((("match_phrase_prefix query")))the `match_phrase_prefix` query:

[source,js]

{ "match_phrase_prefix" : { "brand" : "johnnie walker bl" }

}

// SENSE: 130_Partial_Matching/20_Match_phrase_prefix.json

This query behaves in the same way as the `match_phrase` query, except that it treats the last word in the query string as a prefix. In other words, the preceding example would look for the following:

- `johnnie`
- Followed by `walker`
- Followed by words beginning with `bl`

If you were to run this query through the `validate-query` API, it would produce this explanation:

```
"johnnie walker bl*"
```

Like the `match_phrase` query, it accepts a `slop` parameter (see <>) to make the word order and relative positions ((("slop parameter", "match_prhase_prefix query")))((("match_phrase_prefix query", "slop parameter")))somewhat less rigid:

[source,js]

{ "match_phrase_prefix" : { "brand" : { "query": "walker johnnie bl", <1> "slop": 10 } }

}

// SENSE: 130_Partial_Matching/20_Match_phrase_prefix.json

<1> Even though the words are in the wrong order, the query still matches because we have set a high enough `slop` value to allow some flexibility in word positions.

However, it is always only the last word in the query string that is treated as a prefix.

Earlier, in <>, we warned about the perils of the prefix--how `prefix` queries can be resource intensive. The same is true in

this case.((("match_phrase_prefix query", "caution with"))) A prefix of `a` could match hundreds of thousands of terms. Not only would matching on this many terms be resource intensive, but it would also not be useful to the user.

We can limit the impact ((("match_phrase_prefix query", "max_expansions")))((("max_expansions parameter")))of the prefix expansion by setting `max_expansions` to a reasonable number, such as 50:

[source,js]

{ "match_phrase_prefix" : { "brand" : { "query": "johnnie walker bl", "max_expansions": 50 } }

}

// SENSE: 130_Partial_Matching/20_Match_phrase_prefix.json

The `max_expansions` parameter controls how many terms the prefix is allowed to match. It will find the first term starting with `bl` and keep collecting terms (in alphabetical order) until it either runs out of terms with prefix `bl`, or it has more terms than `max_expansions`.

Don't forget that we have to run this query every time the user types another character, so it needs to be fast. If the first set of results isn't what users are after, they'll keep typing until they get the results that they want.

=== Index-Time Optimizations

All of the solutions we've talked about so far are implemented at *query time*. ((("index time optimizations")))((("partial matching", "index time optimizations")))They don't require any special mappings or indexing patterns; they simply work with the data that you've already indexed.

The flexibility of query-time operations comes at a cost: search performance. Sometimes it may make sense to move the cost away from the query. In a real- time web application, an additional 100ms may be too much latency to tolerate.

By preparing your data at index time, you can make your searches more flexible and improve performance. You still pay a price: increased index size and slightly slower indexing throughput, but it is a price you pay once at index time, instead of paying it on every query.

Your users will thank you.

=== Ngrams for Partial Matching

As we have said before, `` `You can find only terms that exist in the inverted index.'' Although the `` prefix `,` wildcard `, and` regexp` queries demonstrated that that is not strictly true, it *is* true that doing a single-term lookup is much faster than iterating through the terms list to find matching terms on the fly.((("partial matching", "index time optimizations", "n-grams"))) Preparing your data for partial matching ahead of time will increase your search performance.

Preparing your data at index time means choosing the right analysis chain, and the tool that we use for partial matching is the *n-gram*.((("n-grams"))) An n-gram can be best thought of as a *moving window on a word*. The *n* stands for a length. If we were to n-gram the word `quick`, the results would depend on the length we have chosen:

[horizontal]

- Length 1 (unigram): [`q`, `u`, `i`, `c`, `k`]
- Length 2 (bigram): [`qu`, `ui`, `ic`, `ck`]
- Length 3 (trigram): [`qui`, `uic`, `ick`]
- Length 4 (four-gram): [`quic`, `uick`]
- Length 5 (five-gram): [`quick`]

Plain n-grams are useful for matching *somewhere within a word*, a technique that we will use in <>. However, for search-as-you-type, we use a specialized form of n-grams called *edge n-grams*. (((("edge n-grams"))) Edge n-grams are anchored to the beginning of the word. Edge n-gramming the word `quick` would result in this:

- `q`
- `qu`
- `qui`
- `quic`
- `quick`

You may notice that this conforms exactly to the letters that a user searching for ``quick'' would type. In other words, these are the perfect terms to use for instant search!

=== Index-Time Search-as-You-Type

The first step to setting up index-time search-as-you-type is to(((("search-as-you-type", "index time")))((("partial matching", "index time search-as-you-type"))) define our analysis chain, which we discussed in <>, but we will go over the steps again here.

==== Preparing the Index

The first step is to configure a ((("partial matching", "index time search-as-you-type", "preparing the index")))custom `edge_ngram` token filter,(((edge_ngram token filter))) which we will call the `autocomplete_filter`:

[source,js]

{ "filter": { "autocomplete_filter": { "type": "edge_ngram", "min_gram": 1, "max_gram": 20 } }

}

This configuration says that, for any term that this token filter receives, it should produce an n-gram anchored to the start of the word of minimum length 1 and maximum length 20.

Then we need to use this token filter in a custom analyzer,(((analyzers", "autocomplete custom analyzer"))) which we will call the `autocomplete` analyzer:

[source,js]

{ "analyzer": { "autocomplete": { "type": "custom", "tokenizer": "standard", "filter": ["lowercase", "autocomplete_filter" <1>] } }

}

<1> Our custom edge-ngram token filter

This analyzer will tokenize a string into individual terms by using the `standard` tokenizer, lowercase each term, and then produce edge n-grams of each term, thanks to our `autocomplete_filter`.

The full request to create the index and instantiate the token filter and analyzer looks like this:

[source,js]

PUT /my_index { "settings": { "number_of_shards": 1, <1> "analysis": { "filter": { "autocomplete_filter": { <2> "type": "edge_ngram", "min_gram": 1, "max_gram": 20 } }, "analyzer": { "autocomplete": { "type": "custom", "tokenizer": "standard", "filter": ["lowercase", "autocomplete_filter" <3>] } } } }

}

// SENSE: 130_Partial_Matching/35_Search_as_you_type.json

<1> See <>.

<2> First we define our custom token filter.

<3> Then we use it in an analyzer.

You can test this new analyzer to make sure it is behaving correctly by using the `analyze` API:

[source,js]

GET /my_index/_analyze?analyzer=autocomplete

quick brown

// SENSE: 130_Partial_Matching/35_Search_as_you_type.json

The results show us that the analyzer is working correctly. It returns these terms:

- `q`
- `qu`
- `qui`
- `quic`
- `quick`
- `b`
- `br`
- `bro`
- `brow`
- `brown`

To use the analyzer, we need to apply it to a field, which we can do with(("update-mapping API, applying custom autocomplete analyzer to a field"))) the `update-mapping` API:

[source,js]

PUT /my_index/_mapping/my_type { "my_type": { "properties": { "name": { "type": "string", "analyzer": "autocomplete" } } }

}

// SENSE: 130_Partial_Matching/35_Search_as_you_type.json

Now, we can index some test documents:

[source,js]

POST /my_index/my_type/_bulk { "index": { "_id": 1 }} { "name": "Brown foxes" } { "index": { "_id": 2 }}

{ "name": "Yellow furballs" }

// SENSE: 130_Partial_Matching/35_Search_as_you_type.json

==== Querying the Field

If you test out a query for `` `brown fo'' by using ((("partial matching", "index time search-as-you-type", "querying the field")))a simple `` match` query

[source,js]

GET /my_index/my_type/_search { "query": { "match": { "name": "brown fo" } }

}

// SENSE: 130_Partial_Matching/35_Search_as_you_type.json

you will see that *both* documents match, even though the `Yellow furballs` doc contains neither `brown` nor `fo`:

[source,js]

{

"hits": [{ "_id": "1", "_score": 1.5753809, "_source": { "name": "Brown foxes" } }, { "_id": "2", "_score": 0.012520773, "_source": { "name": "Yellow furballs" } }]

}

As always, the `validate-query` API shines some light:

[source,js]

GET /my_index/my_type/_validate/query?explain { "query": { "match": { "name": "brown fo" } }

}

// SENSE: 130_Partial_Matching/35_Search_as_you_type.json

The `explanation` shows us that the query is looking for edge n-grams of every word in the query string:

```
name:b name:br name:bro name:brow name:brown name:f name:fo
```

The `name:f` condition is satisfied by the second document because `furballs` has been indexed as `f`, `fu`, `fur`, and so forth. In retrospect, this is not surprising. The same `autocomplete` analyzer is being applied both at index time and at search time, which in most situations is the right thing to do. This is one of the few occasions when it makes sense to break this rule.

We want to ensure that our inverted index contains edge n-grams of every word, but we want to match only the full words that the user has entered (`brown` and `fo`). ((("analyzers", "changing search analyzer from index analyzer"))) We can do this by using the `autocomplete` analyzer at index time and the `standard` analyzer at search time. One way to change the search analyzer is just to specify it in the query:

[source,js]

GET /my_index/my_type/_search { "query": { "match": { "name": { "query": "brown fo", "analyzer": "standard" <1> } } }

}

// SENSE: 130_Partial_Matching/35_Search_as_you_type.json

<1> This overrides the `analyzer` setting on the `name` field.

Alternatively, we can specify ((("search_analyzer parameter")))((("index_analyzer parameter")))the `index_analyzer` and `search_analyzer` in the mapping for the `name` field itself. Because we want to change only the `search_analyzer`, we can update the existing mapping without having to reindex our data:

[source,js]

PUT /my_index/my_type/_mapping { "my_type": { "properties": { "name": { "type": "string", "index_analyzer": "autocomplete", <1> "search_analyzer": "standard" <2> } } }

}

// SENSE: 130_Partial_Matching/35_Search_as_you_type.json

<1> Use the `autocomplete` analyzer at index time to produce edge n-grams of every term.

<2> Use the `standard` analyzer at search time to search only on the terms that the user has entered.

If we were to repeat the `validate-query` request, it would now give us this explanation:

```
name:brown name:fo
```

Repeating our query correctly returns just the `Brown foxes` document.

Because most of the work has been done at index time, all this query needs to do is to look up the two terms `brown` and `fo`, which is much more efficient than the `match_phrase_prefix` approach of having to find all terms beginning with `fo`.

.Completion Suggester

Using edge n-grams for search-as-you-type is easy to set up, flexible, and fast. However, sometimes it is not fast enough. Latency matters, especially when you are trying to provide instant feedback. Sometimes the fastest way of searching is not to search at all.

The http://bit.ly/1IChV5j[completion suggester] in Elasticsearch((("completion suggester"))) takes a completely different approach. You feed it a list of all possible completions, and it builds them into a *finite state transducer*, an(("Finite State Transducer"))) optimized data structure that resembles a big graph. To search for suggestions, Elasticsearch starts at the beginning of the graph and moves character by character along the matching path. Once it has run out of user input, it looks at all possible endings of the current path to produce a list of suggestions.

This data structure lives in memory and makes prefix lookups extremely fast, much faster than any term-based query could be. It is an excellent match for autocompletion of names and brands, whose words are usually organized in a common order: `Johnny Rotten'' rather than` Rotten Johnny."

When word order is less predictable, edge n-grams can be a better solution than the completion suggester. This particular cat may be skinned in myriad ways.

==== Edge n-grams and Postcodes

The edge n-gram approach can((("postcodes (UK), partial matching with", "using edge n-grams")))(("edge n-grams", "and postcodes"))) also be used for structured data, such as the postcodes example from <>. Of course, the `postcode` field would need to be `analyzed` instead of `not_analyzed`, but you could use the `keyword` tokenizer(("keyword tokenizer", "using for values treated as not_analyzed")))(("not_analyzed fields", "using keyword tokenizer with"))) to treat the postcodes as if they were `not_analyzed`.

[TIP]

The `keyword` tokenizer is the no-operation tokenizer, the tokenizer that does nothing. Whatever string it receives as input, it emits exactly the same string as a single token. It can therefore be used for values that we would normally treat as `not_analyzed` but that require some other analysis transformation such as lowercasing.

==========

This example uses the `keyword` tokenizer to convert the postcode string into a token stream, so that we can use the edge n-gram token filter:

[source,js]

{ "analysis": { "filter": { "postcode_filter": { "type": "edge_ngram", "min_gram": 1, "max_gram": 8 } }, "analyzer": { "postcode_index": { <1> "tokenizer": "keyword", "filter": ["postcode_filter"] }, "postcode_search": { <2> "tokenizer": "keyword" } } }

}

// SENSE: 130_Partial_Matching/35_Postcodes.json

<1> The `postcode_index` analyzer would use the `postcode_filter` to turn postcodes into edge n-grams.

<2> The `postcode_search` analyzer would treat search terms as if they were `not_indexed` .

[[ngrams-compound-words]] === Ngrams for Compound Words

Finally, let's take a look at how n-grams can be used to search languages with compound words. ((("languages", "using many compound words, indexing of")))((("n-grams", "using with compound words")))((("partial matching", "using n-grams for compound words")))((("German", "compound words in"))) German is famous for combining several small words into one massive compound word in order to capture precise or complex meanings. For example:

Aussprachewörterbuch:: Pronunciation dictionary

Militärgeschichte:: Military history

Weißkopfseeadler:: White-headed sea eagle, or bald eagle

Weltgesundheitsorganisation:: World Health Organization

Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz:: The law concerning the delegation of duties for the supervision of cattle marking and the labeling of beef

Somebody searching for `Wörterbuch'' (dictionary) would probably expect to see` Aussprachewörtebuch" in the results list. Similarly, a search for `Adler'' (eagle) should include` Weißkopfseeadler."

One approach to indexing languages like this is to break compound words into their constituent parts using the http://bit.ly/1ygdjjC[compound word token filter]. However, the quality of the results depends on how good your compound-word dictionary is.

Another approach is just to break all words into n-grams and to search for any matching fragments--the more fragments that match, the more relevant the document.

Given that an n-gram is a moving window on a word, an n-gram of any length will cover all of the word. We want to choose a length that is long enough to be meaningful, but not so long that we produce far too many unique terms. A *trigram* (length 3) is ((("trigrams")))probably a good starting point:

[source,js]

PUT /my_index { "settings": { "analysis": { "filter": { "trigrams_filter": { "type": "ngram", "min_gram": 3, "max_gram": 3 } }, "analyzer": { "trigrams": { "type": "custom", "tokenizer": "standard", "filter": ["lowercase", "trigrams_filter"] } } } }, "mappings": { "my_type": { "properties": { "text": { "type": "string", "analyzer": "trigrams" <1> } } } }

}

// SENSE: 130_Partial_Matching/40_Compound_words.json

<1> The `text` field uses the `trigrams` analyzer to index its contents as n-grams of length 3.

Testing the trigrams analyzer with the `analyze` API

[source,js]

GET /my_index/_analyze?analyzer=trigrams

Weißkopfseeadler

// SENSE: 130_Partial_Matching/40_Compound_words.json

returns these terms:

```
wei, eiß, ißk, ßko, kop, opf, pfs, fse, see, eea,ead, adl, dle, ler
```

We can index our example compound words to test this approach:

[source,js]

POST /my_index/my_type/_bulk { "index": { "_id": 1 }} { "text": "Aussprachewörterbuch" } { "index": { "_id": 2 }} { "text": "Militärgeschichte" } { "index": { "_id": 3 }} { "text": "Weißkopfseeadler" } { "index": { "_id": 4 }} { "text": "Weltgesundheitsorganisation" } { "index": { "_id": 5 }}

{ "text": "Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz" }

// SENSE: 130_Partial_Matching/40_Compound_words.json

A search for `` `Adler'' (eagle) becomes a query for the three terms `` adl `, ` dle `, and ` ler``:

[source,js]

GET /my_index/my_type/_search { "query": { "match": { "text": "Adler" } }

}

// SENSE: 130_Partial_Matching/40_Compound_words.json

which correctly matches ``Weißkopfsee-**adler**'':

[source,js]

{ "hits": [{ "_id": "3", "_score": 3.3191128, "_source": { "text": "Weißkopfseeadler" } }]

}

// SENSE: 130_Partial_Matching/40_Compound_words.json

A similar query for `` `Gesundheit'' (health) correctly matches `` Welt-**gesundheit**-sorganisation,'' but it also matches `` `Militär-__ges__-chichte'' and `` Rindfleischetikettierungsüberwachungsaufgabenübertragungs-**ges**-etz,'' both of which also contain the trigram `` ges ``.

Judicious use of the `` minimum_should_match `` parameter can remove these spurious results by requiring that a minimum number of trigrams must be present for a document to be considered a match:

[source,js]

GET /my_index/my_type/_search { "query": { "match": { "text": { "query": "Gesundheit", "minimum_should_match": "80%" } } }

}

// SENSE: 130_Partial_Matching/40_Compound_words.json

This is a bit of a shotgun approach to full-text search and can result in a large inverted index, but it is an effective generic way of indexing languages that use many compound words or that don't use whitespace between words, such as Thai.

This technique is used to increase *recall*—the number of relevant documents that a search returns. It is usually used in combination with other techniques, such as shingles (see <>) to improve precision and the relevance score of each document.

[[controlling-relevance]] == Controlling Relevance

Databases that deal purely in structured data (such as dates, numbers, and string enums) have it easy: they(((("relevance", "controlling"))) just have to check whether a document (or a row, in a relational database) matches the query.

While Boolean yes/no matches are an essential part of full-text search, they are not enough by themselves. Instead, we also need to know how relevant each document is to the query. Full-text search engines have to not only find the matching documents, but also sort them by relevance.

Full-text relevance ((("similarity algorithms")))formulae, or *similarity algorithms*, combine several factors to produce a single relevance `_score` for each document. In this chapter, we examine the various moving parts and discuss how they can be controlled.

Of course, relevance is not just about full-text queries; it may need to take structured data into account as well. Perhaps we are looking for a vacation home with particular features (air-conditioning, sea view, free WiFi). The more features that a property has, the more relevant it is. Or perhaps we want to factor in sliding scales like recency, price, popularity, or distance, while still taking the relevance of a full-text query into account.

All of this is possible thanks to the powerful scoring infrastructure available in Elasticsearch.

We will start by looking at the theoretical side of how Lucene calculates relevance, and then move on to practical examples of how you can control the process.

[[scoring-theory]] === Theory Behind Relevance Scoring

Lucene (and thus Elasticsearch) uses the http://en.wikipedia.org/wiki/Standard_Boolean_model[_Boolean model*] to find matching documents,((("relevance scores", "theory behind", id="ix_relscore", range="startofrange")))(("Boolean Model"))) and a formula called the <<practical-scoring-function,_practical scoring function>>* to calculate relevance. This formula borrows concepts from http://en.wikipedia.org/wiki/Tfidf[_term frequency/inverse document frequency*] *and the* *http://en.wikipedia.org/wiki/Vector_space_model[_vector space model*] but adds more-modern features like a coordination factor, field length normalization, and term or query clause boosting.

[NOTE]

Don't be alarmed! These concepts are not as complicated as the names make them appear. While this section mentions algorithms, formulae, and mathematical models, it is intended for consumption by mere humans. Understanding the algorithms themselves is not as important as understanding the factors that

influence the outcome.

[[boolean-model]] ==== Boolean Model

The *Boolean model* simply applies the `AND`, `OR`, and `NOT` conditions expressed in the query to find all the documents that match.((("and operator")))(("not operator")))(("or operator"))) A query for

```
full AND text AND search AND (elasticsearch OR lucene)
```

will include only documents that contain all of the terms `full`, `text`, and `search`, and either `elasticsearch` or `lucene`.

This process is simple and fast. It is used to exclude any documents that cannot possibly match the query.

[[tfidf]] ==== Term Frequency/Inverse Document Frequency (TF/IDF)

Once we have a list of matching documents, they need to be ranked by relevance.((("Term Frequency/Inverse Document Frequency (TF/IDF) similarity algorithm"))) Not all documents will contain all the terms, and some terms are more important than others. The relevance score of the whole document depends (in part) on the *weight* of each query term that appears in that document.

The weight of a term is determined by three factors, which we already introduced in <>. The formulae are included for interest's sake, but you are not required to remember them.

[[tf]] ===== Term frequency

How often does the term appear in this document?(((("Term Frequency/Inverse Document Frequency (TF/IDF) similarity algorithm", "term frequency"))) The more often, the *higher* the weight. A field containing five mentions of the same term is more likely to be relevant than a field containing just one mention. The term frequency is calculated as follows:

..... tf(t in d) = √frequency <1> .....

<1> The term frequency (`tf`) for term `t` in document `d` is the square root of the number of times the term appears in the document.

If you don't care about how often a term appears in a field, and all you care about is that the term is present, then you can disable term frequencies in the field mapping:

[source,json]

PUT /my_index { "mappings": { "doc": { "properties": { "text": { "type": "string", "index_options": "docs" <1> } } } }

}

<1> Setting `index_options` to `docs` will disable term frequencies and term positions. A field with this mapping will not count how many times a term appears, and will not be usable for phrase or proximity queries. Exact-value `not_analyzed` string fields use this setting by default.

[[idf]] ===== Inverse document frequency

How often does the term appear in all documents in the collection? The more often, the *lower* the weight.((("inverse document frequency")))((("Term Frequency/Inverse Document Frequency (TF/IDF) similarity algorithm", "inverse document frequency"))) Common terms like `and` or `the` contribute little to relevance, as they appear in most documents, while uncommon terms like `elastic` or `hippopotamus` help us zoom in on the most interesting documents. The inverse document frequency is calculated as follows:

..... idf(t) = 1 + log (numDocs / (docFreq + 1)) <1> .....

<1> The inverse document frequency (`idf`) of term `t` is the logarithm of the number of documents in the index, divided by the number of documents that contain the term.

[[field-norm]] ===== Field-length norm

How long is the field? ((("Term Frequency/Inverse Document Frequency (TF/IDF) similarity algorithm", "field-length norm")))((("field-length norm")))The shorter the field, the *higher* the weight. If a term appears in a short field, such as a `title` field, it is more likely that the content of that field is *about* the term than if the same term appears in a much bigger `body` field. The field length norm is calculated as follows:

..... norm(d) = 1 / √numTerms <1> .....

<1> The field-length norm (`norm`) is the inverse square root of the number of terms in the field.

While the field-length ((("string fields", "field-length norm")))norm is important for full-text search, many other fields don't need norms. Norms consume approximately 1 byte per `string` field per document in the index, whether or not a document contains the field. Exact-value `not_analyzed` string fields have norms disabled by default, but you can use the field mapping to disable norms on `analyzed` fields as well:

[source,json]

PUT /my_index { "mappings": { "doc": { "properties": { "text": { "type": "string", "norms": { "enabled": false } <1> } } } }

}

<1> This field will not take the field-length norm into account. A long field and a short field will be scored as if they were the same length.

For use cases such as logging, norms are not useful. All you care about is whether a field contains a particular error code or a particular browser identifier. The length of the field does not affect the outcome. Disabling norms can save a significant amount of memory.

===== Putting it together

These three factors--term frequency, inverse document frequency, and field-length norm--are calculated and stored at index time.((("weight", "calculation of"))) Together, they are used to calculate the *weight* of a single term in a particular document.

[TIP]

When we refer to *documents* in the preceding formulae, we are actually talking about a field within a document. Each field has its own inverted index and thus, for TF/IDF purposes, the value of the field is the value of the document.

==========

When we run a simple `term` query with `explain` set to `true` (see <>), you will see that the only factors involved in calculating the score are the ones explained in the preceding sections:

[role="pagebreak-before"]

[source,json]

PUT /my_index/doc/1 { "text" : "quick brown fox" }

GET /my_index/doc/_search?explain { "query": { "term": { "text": "fox" } }

}

The (abbreviated) `explanation` from the preceding request is as follows:

..... weight(text:fox in 0) [PerFieldSimilarity]: 0.15342641 <1> result of: fieldWeight in 0 0.15342641 product of: tf(freq=1.0), with freq of 1: 1.0 <2> idf(docFreq=1, maxDocs=1): 0.30685282 <3> fieldNorm(doc=0): 0.5 <4> .....

<1> The final `score` for term `fox` in field `text` in the document with internal Lucene doc ID `0`.

<2> The term `fox` appears once in the `text` field in this document.

<3> The inverse document frequency of `fox` in the `text` field in all documents in this index.

<4> The field-length normalization factor for this field.

Of course, queries usually consist of more than one term, so we need a way of combining the weights of multiple terms. For this, we turn to the vector space model.

[[vector-space-model]] ==== Vector Space Model

The *vector space model* provides a way of (((("Vector Space Model")))comparing a multiterm query against a document. The output is a single score that represents how well the document matches the query. In order to do this, the model represents both the document and the query as *vectors*.

A vector is really just a one-dimensional array containing numbers, for example:

```
[1,2,5,22,3,8]
```

In the vector space(((("Term Frequency/Inverse Document Frequency (TF/IDF) similarity algorithm", "in Vector Space Model"))) model, each number in the vector is((("weight", "calculation of", "in Vector Space Model"))) the *weight* of a term, as calculated with <>.

[TIP]

While TF/IDF is the default way of calculating term weights for the vector space model, it is not the only way. Other models

like Okapi-BM25 exist and are available in Elasticsearch. TF/IDF is the default because it is a simple, efficient algorithm that produces high-quality search results and has stood the test of time.

==========

Imagine that we have a query for `happy hippopotamus.'' A common word like happy will have a low weight, while an uncommon term like hippopotamus will have a high weight. Let's assume that happy has a weight of 2 and hippopotamus has a weight of 5. We can plot this simple two-dimensional vector&#x2014; [2,5]`—as a line on a graph starting at point (0,0) and ending at point (2,5), as shown in <>.

[[img-vector-query]] .A two-dimensional query vector for ``happy hippopotamus'' represented image::images/elas_17in01.png["The query vector plotted on a graph"]

Now, imagine we have three documents:

1. I am *happy* in summer.
2. After Christmas I'm a *hippopotamus*.
3. The *happy hippopotamus* helped Harry.

We can create a similar vector for each document, consisting of the weight of each query term— `happy` and `hippopotamus` —that appears in the document, and plot these vectors on the same graph, as shown in <>:

- Document 1: `(happy,_____)` — `[2,0]`
- Document 2: `( ___ ,hippopotamus)` — `[0,5]`
- Document 3: `(happy,hippopotamus)` — `[2,5]`

[[img-vector-docs]] .Query and document vectors for ``happy hippopotamus'' image::images/elas_17in02.png["The query and document vectors plotted on a graph"]

The nice thing about vectors is that they can be compared. By measuring the angle between the query vector and the document vector, it is possible to assign a relevance score to each document. The angle between document 1 and the query is large, so it is of low relevance. Document 2 is closer to the query, meaning that it is reasonably relevant, and document 3 is a perfect match.

[TIP]

In practice, only two-dimensional vectors (queries with two terms) can be plotted easily on a graph. Fortunately, *linear algebra*—the branch of mathematics that deals with vectors--provides tools to compare the angle between multidimensional vectors, which means that we can apply the same principles explained above to queries that consist of many terms.

You can read more about how to compare two vectors by using http://en.wikipedia.org/wiki/Cosine_similarity[_cosine similarity_].

==========

Now that we have talked about the theoretical basis of scoring, we can move on to see how scoring is implemented in Lucene.((("relevance scores", "theory behind", range="endofrange", startref="ix_relscore")))

[[practical-scoring-function]] === Lucene's Practical Scoring Function

For multiterm queries, Lucene takes(((("relevance", "controlling", "Lucene's practical scoring function", id="ix_relcontPCF", range="startofrange")))((("Boolean Model"))) the <>, <>, and the <> and combines (((("Term Frequency/Inverse Document Frequency (TF/IDF) similarity algorithm"))))((("Vector Space Model"))) them in a single efficient package that collects matching documents and scores them as it goes.

A multiterm query like

[source,json]

GET /my_index/doc/_search { "query": { "match": { "text": "quick fox" } }

}

is rewritten internally to look like this:

[source,json]

GET /my_index/doc/_search { "query": { "bool": { "should": [{"term": { "text": "quick" }}, {"term": { "text": "fox" }}] } }

}

The `bool` query implements the Boolean model and, in this example, will include only documents that contain either the term `quick` or the term `fox` or both.

As soon as a document matches a query, Lucene calculates its score for that query, combining the scores of each matching term. The formula used for scoring is called the *practical scoring function*.((("practical scoring function"))) It looks intimidating, but don't be put off--most of the components you already know. It introduces a few new elements that we discuss next.

..... score(q,d) = <1> queryNorm(q) <2> · coord(q,d) <3> · ∑ (<4> tf(t in d) <5> · idf(t)² <6> · t.getBoost() <7> · norm(t,d) <8>) (t in q) <4> .....

<1> `score(q,d)` is the relevance score of document `d` for query `q`.

<2> `queryNorm(q)` is the <> (new).

<3> `coord(q,d)` is the <> (new).

<4> The sum of the weights for each term `t` in the query `q` for document `d`.

<5> `tf(t in d)` is the <> for term `t` in document `d`.

<6> `idf(t)` is the <> for term `t`.

<7> `t.getBoost()` is the <> that has been applied to the query (new).

<8> `norm(t,d)` is the <>, combined with the <>, if any. (new).

You should recognize `score`, `tf`, and `idf`. The `queryNorm`, `coord`, `t.getBoost`, and `norm` are new.

We will talk more about <> later in this chapter, but first let's get query normalization, coordination, and index-time field-level boosting out of the way.

[[query-norm]] ==== Query Normalization Factor

The *query normalization factor* (`queryNorm`) is (((("practical scoring function", "query normalization factor")))(((("query normalization factor")))(((("normalization", "query normalization factor")))an attempt to *normalize* a query so that the results from one query may be compared with the results of another.

[TIP]

Even though the intent of the query norm is to make results from different queries comparable, it doesn't work very well. The only purpose of the relevance `_score` is to sort the results of the current query in the correct order. You should not try to compare the relevance scores from different queries.

==========

This factor is calculated at the beginning of the query. The actual calculation depends on the queries involved, but a typical implementation is as follows:

..... queryNorm = 1 / √sumOfSquaredWeights <1> .....

<1> The `sumOfSquaredWeights` is calculated by adding together the IDF of each term in the query, squared.

TIP: The same query normalization factor is applied to every document, and you have no way of changing it. For all intents and purposes, it can be ignored.

[[coord]] ==== Query Coordination

The *coordination factor* (`coord`) is used to((("coordination factor (coord)")))(((("query coordination")))(((("practical scoring function", "coordination factor"))) reward documents that contain a higher percentage of the query terms. The more query terms that appear in the document, the greater the chances that the document is a good match for the query.

Imagine that we have a query for `quick brown fox` , and that the weight for each term is 1.5. Without the coordination factor, the score would just be the sum of the weights of the terms in a document. For instance:

- Document with `fox` -> score: 1.5
- Document with `quick fox` -> score: 3.0
- Document with `quick brown fox` -> score: 4.5

The coordination factor multiplies the score by the number of matching terms in the document, and divides it by the total number of terms in the query. With the coordination factor, the scores would be as follows:

- Document with `fox` -> score: `1.5 * 1 / 3` = 0.5
- Document with `quick fox` -> score: `3.0 * 2 / 3` = 2.0
- Document with `quick brown fox` -> score: `4.5 * 3 / 3` = 4.5

The coordination factor results in the document that contains all three terms being much more relevant than the document that contains just two of them.

Remember that the query for `quick brown fox` is rewritten into a `bool` query like this:

[source,json]

GET /_search { "query": { "bool": { "should": [{ "term": { "text": "quick" }}, { "term": { "text": "brown" }}, { "term": { "text": "fox" }}] } }

}

The `bool` query uses query coordination by default for all `should` clauses, but it does allow you to disable coordination. Why might you want to do this? Well, usually the answer is, you don't. Query coordination is usually a good thing. When you use a `bool` query to wrap several high-level queries like the `match` query, it also makes sense to leave coordination enabled. The more clauses that match, the higher the degree of overlap between your search request and the documents that are returned.

However, in some advanced use cases, it might make sense to disable coordination. Imagine that you are looking for the synonyms `jump`, `leap`, and `hop`. You don't care how many of these synonyms are present, as they all represent the same concept. In fact, only one of the synonyms is likely to be present. This would be a good case for disabling the coordination factor:

[source,json]

GET /_search { "query": { "bool": { "disable_coord": true, "should": [{ "term": { "text": "jump" }}, { "term": { "text": "hop" }}, { "term": { "text": "leap" }}] } }

}

When you use synonyms (see <>), this is exactly what happens internally: the rewritten query disables coordination for the synonyms. ((("synonyms", "query coordination and"))) Most use cases for disabling coordination are handled automatically; you don't need to worry about it.

[[index-boost]] ==== Index-Time Field-Level Boosting

We will talk about *boosting* a field--making it ((("indexing", "field-level index time boosts")))((("boosting", "index time field-level boosting")))((("practical scoring function", "index time field-level boosting")))more important than other fields--at query time in <>. It is also possible to apply a boost to a field at index time. Actually, this boost is applied to every term in the field, rather than to the field itself.

To store this boost value in the index without using more space than necessary, this field-level index-time boost is combined with the (((“field-length norm")))field-length norm (see <>) and stored in the index as a single byte. This is the value returned by `norm(t,d)` in the preceding formula.

[WARNING]

We strongly recommend against using field-level index-time boosts for a few reasons:

- Combining the boost with the field-length norm and storing it in a single byte means that the field-length norm loses precision. The result is that Elasticsearch is unable to distinguish between a field containing three words and a field containing five words.
- To change an index-time boost, you have to reindex all your documents. A query-time boost, on the other hand, can be changed with every query.
- If a field with an index-time boost has multiple values, the boost is multiplied by itself for every value, dramatically increasing the weight for that field.

<> is a much simpler, cleaner, more flexible option.

==========

With query normalization, coordination, and index-time boosting out of the way, we can now move on to the most useful tool for influencing the relevance calculation: query-time boosting.(((“relevance", "controlling", "Lucene's practical scoring function", range="endofrange", startref="ix_relcontPCF")))

[[query-time-boosting]] === Query-Time Boosting

In <>, we explained (((("relevance", "controlling", "query time boosting")))((("boosting", "query-time")))how you could use the `boost` parameter at search time to give one query clause more importance than another. For instance:

[source,json]

GET /_search { "query": { "bool": { "should": [{ "match": { "title": { "query": "quick brown fox", "boost": 2 <1> } } }, { "match": { <2> "content": "quick brown fox" } }] } }

}

<1> The `title` query clause is twice as important as the `content` query clause, because it has been boosted by a factor of `2`.

<2> A query clause without a `boost` value has a neutral boost of `1`.

Query-time boosting is the main tool that you can use to tune relevance. Any type of query accepts a `boost` parameter. ((("boost parameter", "setting value"))) Setting a `boost` of `2` doesn't simply double the final `_score`; the actual boost value that is applied goes through normalization and some internal optimization. However, it does imply that a clause with a boost of `2` is twice as important as a clause with a boost of `1`.

Practically, there is no simple formula for deciding on the `correct'' boost value for a particular query clause. It's a matter of try-it-and-see. Remember that `boost` is just one of the factors involved in the relevance score; it has to compete with the other factors. For instance, in the preceding example, the `title` field will probably already have a` natural" boost over the `content` field thanks ((("field-length norm")))to the <> (titles are usually shorter than the related content), so don't blindly boost fields just because you think they should be boosted. Apply a boost and check the results. Change the boost and check again.

==== Boosting an Index

When searching across multiple indices, you((("boosting", "query-time", "boosting an index")))((("indices", "boosting an index"))) can boost an entire index over the others with the `indices_boost` parameter.((("indices_boost parameter"))) This could be used, as in the next example, to give more weight to documents from a more recent index:

[source,json]

GET /docs*2014*/_search <1> { "indices_boost": { <2> "docs_2014_10": 3, "docs_2014_09": 2 }, "query": { "match": { "text": "quick brown fox" } }

}

<1> This multi-index search covers all indices beginning with `docs_2014_`.

<2> Documents in the `docs_2014_10` index will be boosted by `3`, those in `docs_2014_09` by `2`, and any other matching indices will have a neutral boost of `1`.

==== t.getBoost()

These boost values are represented in the <> by the `t.getBoost()` element.((("practical scoring function", "t.getBoost() method")))((("boosting", "query-time", "t.getBoost()")))((("t.getBoost() method"))) Boosts are not applied at the level that they appear in the query DSL. Instead, any boost values are combined and passsed down to the individual terms. The `t.getBoost()` method returns any `boost` value applied to the term itself or to any of the queries higher up the chain.

[TIP]

In fact, reading the <> output is a little more complex than that. You won't see the `boost` value or `t.getBoost()` mentioned in the `explanation` at all. Instead, the boost is rolled into the <> that is applied to a particular term. Although we said that the `queryNorm` is the same for every term, you will see that the `queryNorm` for a boosted term is higher than the `queryNorm` for an unboosted term.

==========

[[query-scoring]] === Manipulating Relevance with Query Structure

The Elasticsearch query DSL is immensely flexible.((("relevance", "controlling", "manipulating relevance with query structure")))((("queries", "manipulating relevance with query structure"))) You can move individual query clauses up and down the query hierarchy to make a clause more or less important. For instance, imagine the following query:

```
quick OR brown OR red OR fox
```

We could write this as a `bool` query with (((("bool query", "manipulating relevance with query structure")))all terms at the same level:

[source,json]

GET /_search { "query": { "bool": { "should": [{ "term": { "text": "quick" }}, { "term": { "text": "brown" }}, { "term": { "text": "red" }}, { "term": { "text": "fox" }}] } }

}

But this query might score a document that contains `quick`, `red`, and `brown` the same as another document that contains `quick`, `red`, and `fox`. *Red* and *brown* are synonyms and we probably only need one of them to match. Perhaps we really want to express the query as follows:

```
quick OR (brown OR red) OR fox
```

According to standard Boolean logic, this is exactly the same as the original query, but as we have already seen in <>, a `bool` query does not concern itself only with whether a document matches, but also with how *well* it matches.

A better way to write this query is as follows:

[source,json]

GET /_search { "query": { "bool": { "should": [{ "term": { "text": "quick" }}, { "term": { "text": "fox" }}, { "bool": { "should": [{ "term": { "text": "brown" }}, { "term": { "text": "red" }}] } }] } }

}

Now, `red` and `brown` compete with each other at their own level, and `quick`, `fox`, and `red OR brown` are the top-level competitive terms.

We have already discussed how the <>, <>, <>, <>, and <> queries can be used to manipulate scoring. In the rest of this chapter, we present three other scoring-related queries: the `boosting` query, the `constant_score` query, and the `function_score` query.

[[not-quite-not]] === Not Quite Not

A search on the Internet for `` `Apple'' is likely to return results about the company, the fruit, ((("relevance", "controlling", "must_not clause in bool query")))((("bool query", "must_not clause")))and various recipes. We could try to narrow it down to just the company by excluding words like `` pie `,` tart `,` crumble `, and` tree `, using a` must_not `clause in a` bool` query:

[source,json]

GET /_search { "query": { "bool": { "must": { "match": { "text": "apple" } }, "must_not": { "match": { "text": "pie tart fruit crumble tree" } } } }

}

But who is to say that we wouldn't miss a very relevant document about Apple the company by excluding `tree` or `crumble`? Sometimes, `must_not` can be too strict.

[[boosting-query]] ==== boosting Query

The http://bit.ly/1IO281f[`boosting` query] solves(((("boosting query")))((("relevance", "controlling", "boosting query"))) this problem. It allows us to still include results that appear to be about the fruit or the pastries, but to downgrade them--to rank them lower than they would otherwise be:

[source,json]

GET /_search { "query": { "boosting": { "positive": { "match": { "text": "apple" } }, "negative": { "match": { "text": "pie tart fruit crumble tree" } }, "negative_boost": 0.5 } }

}

It accepts a `positive` query and a `negative` query.((("positive query and negative query (in boosting query"))) Only documents that match the `positive` query will be included in the results list, but documents that also match the `negative` query will be downgraded by multiplying the original `_score` of((("negative_boost"))) the document with the `negative_boost`.

For this to work, the `negative_boost` must be less than `1.0`. In this example, any documents that contain any of the negative terms will have their `_score` cut in half.

[[ignoring-tfidf]] === Ignoring TF/IDF

Sometimes we just don't care about TF/IDF.((("relevance", "controlling", "ignoring TF/IDF")))((("Term Frequency/Inverse Document Frequency (TF/IDF) similarity algorithm", "ignoring"))) All we want to know is that a certain word appears in a field. Perhaps we are searching for a vacation home and we want to find houses that have as many of these features as possible:

- WiFi
- Garden
- Pool

The vacation home documents look something like this:

[source,json]

{ "description": "A delightful four-bedroomed house with ... " }

We could use a simple `match` query:

[source,json]

GET /_search { "query": { "match": { "description": "wifi garden pool" } }

}

However, this isn't really *full-text search*. In this case, TF/IDF just gets in the way. We don't care whether `wifi` is a common term, or how often it appears in the document. All we care about is that it does appear. In fact, we just want to rank houses by the number of features they have--the more, the better. If a feature is present, it should score `1`, and if it isn't, `0`.

[[constant-score-query]] ==== constant_score Query

Enter the http://bit.ly/1DIgSAK[`constant_score`] query. This (((("constant_score query")))query can wrap either a query or a filter, and assigns a score of `1` to any documents that match, regardless of TF/IDF:

[source,json]

GET /_search { "query": { "bool": { "should": [{ "constant_score": { "query": { "match": { "description": "wifi" } } }, { "constant_score": { "query": { "match": { "description": "garden" } } }, { "constant_score": { "query": { "match": { "description": "pool" } } }] } }

}

Perhaps not all features are equally important--some have more value to the user than others. If the most important feature is the pool, we could boost that clause to make it count for more:

[source,json]

GET /_search { "query": { "bool": { "should": [{ "constant_score": { "query": { "match": { "description": "wifi" } } }, { "constant_score": { "query": { "match": { "description": "garden" } } }, { "constant_score": { "boost": 2 <1> "query": { "match": { "description": "pool" } } }] } }

}

<1> A matching `pool` clause would add a score of `2`, while the other clauses would add a score of only `1` each.

NOTE: The final score for each result is not simply the sum of the scores of all matching clauses. The <> and <> are still taken into account.

We could improve our vacation home documents by adding a `not_analyzed` `features` field to our vacation homes:

[source,json]

{ "features": ["wifi", "pool", "garden"] }

By default, a `not_analyzed` field has <> disabled (("not_analyzed fields", "field length norms and index_options")))and has `index_options` set to `docs`, disabling <>, but the problem remains: the <> of each term is still taken into account.

We could use the same approach that we used previously, with the `constant_score` query:

[source,json]

GET /_search { "query": { "bool": { "should": [{ "constant_score": { "query": { "match": { "features": "wifi" }} }}, { "constant_score": { "query": { "match": { "features": "garden" }} }}, { "constant_score": { "boost": 2 "query": { "match": { "features": "pool" }} }}] } }

}

Really, though, each of these features should be treated like a filter. A vacation home either has the feature or it doesn't--a filter seems like it would be a natural fit. On top of that, if we use filters, we can benefit from filter caching.

The problem is this: filters don't score. What we need is a way of bridging the gap between filters and queries. The `function_score` query does this and a whole lot more.

[[function-score-query]] === function_score Query

The http://bit.ly/1sCKtHW[`function_score` query] is the ultimate tool for taking control of the scoring process. ((("function_score query")))(("relevance", "controlling", "function_score query"))) It allows you to apply a function to each document that matches the main query in order to alter or completely replace the original query `_score`.

In fact, you can apply different functions to *subsets* of the main result set by using filters, which gives you the best of both worlds: efficient scoring with cacheable filters.

It supports several predefined functions out of the box:

`weight`::

```
Apply a simple boost to each document without the boost being
normalized: a `weight` of `2` results in `2 * _score`.
```

`field_value_factor`::

```
Use the value of a field in the document to alter the `_score`,  such as
factoring in a `popularity` count or number of `votes`.
```

`random_score`::

```
Use consistently random scoring to sort results differently for every user,
while maintaining the same sort order for a single user.
```

Decay functions— `linear`, `exp`, `gauss`::

```
Incorporate sliding-scale values like `publish_date`, `geo_location`, or
`price` into the `_score` to prefer recently published documents, documents
near a latitude/longitude (lat/lon) point, or documents near a specified price point.
```

`script_score`::

```
Use a custom script to take complete control of the scoring logic. If your
needs extend beyond those of the functions in this list, write a custom
script to implement the logic that you need.
```

Without the `function_score` query, we would not be able to combine the score from a full-text query with a factor like recency. We would have to sort either by `_score` or by `date`; the effect of one would obliterate the effect of the other. This query allows you to blend the two together: to still sort by full-text relevance, but giving extra weight to recently published documents, or popular documents, or products that are near the user's price point. As you can imagine, a query that supports all of this can look fairly complex. We'll start with a simple use case and work our way up the complexity ladder.

[[boosting-by-popularity]] === Boosting by Popularity

Imagine that we have a website that hosts blog posts and enables users to vote for the blog posts that they like. ((("relevance", "controlling", "boosting by popularity")))((("popularity", "boosting by")))((("boosting", "by popularity"))) We would like more-popular posts to appear higher in the results list, but still have the full-text score as the main relevance driver. We can do this easily by storing the number of votes with each blog post:

[role="pagebreak-before"]

[source,json]

PUT /blogposts/post/1 { "title": "About popularity", "content": "In this post we will talk about...", "votes": 6

}

At search time, we can use the `function_score` query ((("function_score query", "field_value_factor function"))) ((("field_value_factor function")))with the `field_value_factor` function to combine the number of votes with the full-text relevance score:

[source,json]

GET /blogposts/post/_search { "query": { "function_score": { <1> "query": { <2> "multi_match": { "query": "popularity", "fields": ["title", "content"] } }, "field_value_factor": { <3> "field": "votes" <4> } } }

}

<1> The `function_score` query wraps the main query and the function we would like to apply.

<2> The main query is executed first.

<3> The `field_value_factor` function is applied to every document matching the main `query`.

<4> Every document *must* have a number in the `votes` field for the `function_score` to work.

In the preceding example, the final `_score` for each document has been altered as follows:

```
new_score = old_score * number_of_votes
```

This will not give us great results. The full-text `_score` range usually falls somewhere between 0 and 10. As can be seen in <>, a blog post with 10 votes will completely swamp the effect of the full-text score, and a blog post with 0 votes will reset the score to zero.

[[img-popularity-linear]] .Linear popularity based on an original `_score` of `2.0` image::images/elas_1701.png[Linear popularity based on an original `_score` of `2.0`]

==== modifier

A better way to incorporate popularity is to smooth out the `votes` value with some `modifier`. ((("modifier parameter"))) ((("field_value_factor function", "modifier parameter")))In other words, we want the first few votes to count a lot, but for each subsequent vote to count less. The difference between 0 votes and 1 vote should be much bigger than the difference between 10 votes and 11 votes.

A typical `modifier` for this use case is `log1p`, which changes the formula to the following:

```
new_score = old_score * log(1 + number_of_votes)
```

The `log` function smooths out the effect of the `votes` field to provide a curve like the one in <>.

[[img-popularity-log]] .Logarithmic popularity based on an original `_score` of `2.0`
image::images/elas_1702.png[Logarithmic popularity based on an original `_score` of `2.0`]

The request with the `modifier` parameter looks like the following:

[source,json]

GET /blogposts/post/_search { "query": { "function_score": { "query": { "multi_match": { "query": "popularity", "fields": ["title", "content"] } }, "field_value_factor": { "field": "votes", "modifier": "log1p" <1> } } }

}

<1> Set the `modifier` to `log1p`.

[role="pagebreak-before"] The available modifiers are `none` (the default), `log`, `log1p`, `log2p`, `ln`, `ln1p`, `ln2p`, `square`, `sqrt`, and `reciprocal`. You can read more about them in the http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-function-score-query.html#_field_value_factor[`field_value_factor` documentation].

==== factor

The strength of the popularity effect can be increased or decreased by multiplying the value((("factor (function_score)"))) ((("field_value_factor function", "factor parameter"))) in the `votes` field by some number, called the `factor`:

[source,json]

GET /blogposts/post/_search { "query": { "function_score": { "query": { "multi_match": { "query": "popularity", "fields": ["title", "content"] } }, "field_value_factor": { "field": "votes", "modifier": "log1p", "factor": 2 <1> } } }

}

<1> Doubles the popularity effect

Adding in a `factor` changes the formula to this:

```
new_score = old_score * log(1 + factor * number_of_votes)
```

A `factor` greater than `1` increases the effect, and a `factor` less than `1` decreases the effect, as shown in <>.

[[img-popularity-factor]] .Logarithmic popularity with different factors image::images/elas_1703.png[Logarithmic popularity with different factors]

==== boost_mode

Perhaps multiplying the full-text score by the result of the `field_value_factor` function (((("function_score query", "boost_mode parameter")))((("boost_mode parameter")))still has too large an effect. We can control how the result of a function is combined with the `_score` from the query by using the `boost_mode` parameter, which accepts the following values:

`multiply` :: Multiply the `_score` with the function result (default)

`sum` :: Add the function result to the `_score`

`min` :: The lower of the `_score` and the function result

`max` :: The higher of the `_score` and the function result

`replace` :: Replace the `_score` with the function result

If, instead of multiplying, we add the function result to the `_score`, we can achieve a much smaller effect, especially if we use a low `factor`:

[source,json]

GET /blogposts/post/_search { "query": { "function_score": { "query": { "multi_match": { "query": "popularity", "fields": ["title", "content"] } }, "field_value_factor": { "field": "votes", "modifier": "log1p", "factor": 0.1 }, "boost_mode": "sum" <1> } }

}

<1> Add the function result to the `_score`.

The formula for the preceding request now looks like this (see <>):

```
new_score = old_score + log(1 + 0.1 * number_of_votes)
```

[[img-popularity-sum]] .Combining popularity with `sum` image::images/elas_1704.png["Combining popularity with `sum`"]

==== max_boost

Finally, we can cap the maximum effect(((("function_score query", "max_boost parameter")))((("max_boost parameter"))) that the function can have by using the `max_boost` parameter:

[source,json]

GET /blogposts/post/_search { "query": { "function_score": { "query": { "multi_match": { "query": "popularity", "fields": ["title", "content"] } }, "field_value_factor": { "field": "votes", "modifier": "log1p", "factor": 0.1 }, "boost_mode": "sum", "max_boost": 1.5 <1> } }

}

<1> Whatever the result of the `field_value_factor` function, it will never be greater than `1.5`.

NOTE: The `max_boost` applies a limit to the result of the function only, not to the final `_score`.

[[function-score-filters]] === Boosting Filtered Subsets

Let's return to the problem that we were dealing with in <>, where we wanted to score((("boosting", "filtered subsets"))) ((("relevance", "controlling", "boosting filtered subsets"))) vacation homes by the number of features that each home possesses. We ended that section by wishing for a way to use cached filters to affect the score, and with the `function_score` query we can do just that.((("function_score query", "boosting filtered subsets")))

The examples we have shown thus far have used a single function for all documents. Now we want to divide the results into subsets by using filters (one filter per feature), and apply a different function to each subset.

The function that we will use in this example is ((("weight function")))the `weight`, which is similar to the `boost` parameter accepted by any query. The difference is that the `weight` is not normalized by Lucene into some obscure floating-point number; it is used as is.

The structure of the query has to change somewhat to incorporate multiple functions:

[source,json]

GET /_search { "query": { "function_score": { "filter": { <1> "term": { "city": "Barcelona" } }, "functions": [<2> { "filter": { "term": { "features": "wifi" }}, <3> "weight": 1 }, { "filter": { "term": { "features": "garden" }}, <3> "weight": 1 }, { "filter": { "term": { "features": "pool" }}, <3> "weight": 2 <4> }], "score_mode": "sum", <5> } }

}

<1> This `function_score` query has a `filter` instead of a `query`.

<2> The `functions` key holds a list of functions that should be applied.

<3> The function is applied only if the document matches the (optional) `filter`.

<4> The `pool` feature is more important than the others so it has a higher `weight`.

<5> The `score_mode` specifies how the values from each function should be combined.

The new features to note in this example are explained in the following sections.

==== filter Versus query

The first thing to note is that we have specified a `filter` instead ((("filters", "in function_score query")))of a `query`. In this example, we do not need full-text search. We just want to return all documents that have `Barcelona` in the `city` field, logic that is better expressed as a filter instead of a query. All documents returned by the filter will have a `_score` of `1`. The `function_score` query accepts either a `query` or a `filter`. If neither is specified, it will default to using the `match_all` query.

==== functions

The `functions` key holds an array of functions to apply.((("function_score query", "functions key"))) Each entry in the array may also optionally specify a `filter`, in which case the function will be applied only to documents that match that filter. In this example, we apply a `weight` of `1` (or `2` in the case of `pool`) to any document that matches the filter.

==== score_mode

Each function returns a result, and we need a way of reducing these multiple results to a single value that can be combined with the original `_score`. This is the role ((("function_score query", "score_mode parameter")))(((score_mode parameter")))of the `score_mode` parameter, which accepts the following values:

`multiply`::

Function results are multiplied together (default).

`sum` ::
Function results are added up.

`avg` ::
The average of all the function results.

`max` ::
The highest function result is used.

`min` ::
The lowest function result is used.

`first` ::
Uses only the result from the first function that either doesn't have a filter or that has a filter matching the document.

In this case, we want to add the `weight` results from each matching filter together to produce the final score, so we have used the `sum` score mode.

Documents that don't match any of the filters will keep their original `_score` of `1`.

[[random-scoring]] === Random Scoring

You may have been wondering what *consistently random scoring* is, or why you would ever want to use it.((("consistently random scoring")))(("relevance", "controlling", "random scoring"))) The previous example provides a good use case. All results from the previous example would receive a final `_score` of 1, 2, 3, 4, or 5. Maybe there are only a few homes that score 5, but presumably there would be a lot of homes scoring 2 or 3.

As the owner of the website, you want to give your advertisers as much exposure as possible. With the current query, results with the same `_score` would be returned in the same order every time. It would be good to introduce some randomness here, to ensure that all documents in a single score level get a similar amount of exposure.

We want every user to see a different random order, but we want the same user to see the same order when clicking on page 2, 3, and so forth. This is what is meant by *consistently random*.

The `random_score` function, which(("function_score query", "random_score function")))(("random_score function"))) outputs a number between 0 and 1, will produce consistently random results when it is provided with the same `seed` value, such as a user's session ID:

[source,json]

GET /_search { "query": { "function_score": { "filter": { "term": { "city": "Barcelona" } }, "functions": [{ "filter": { "term": { "features": "wifi" }}, "weight": 1 }, { "filter": { "term": { "features": "garden" }}, "weight": 1 }, { "filter": { "term": { "features": "pool" }}, "weight": 2 }, { "random_score": { <1> "seed": "the users session id" <2> } }], "score_mode": "sum", } }

}

<1> The `random_score` clause doesn't have any `filter`, so it will be applied to all documents.

<2> Pass the user's session ID as the `seed`, to make randomization consistent for that user. The same `seed` will result in the same randomization.

Of course, if you index new documents that match the query, the order of results will change regardless of whether you use consistent randomization or not.

[[decay-functions]] === The Closer, The Better

Many variables could influence the user's choice of vacation home.((("relevance", "controlling", "using decay functions"))) Maybe she would like to be close to the center of town, but perhaps would be willing to settle for a place that is a bit farther from the center if the price is low enough. Perhaps the reverse is true: she would be willing to pay more for the best location.

If we were to add a filter that excluded any vacation homes farther than 1 kilometer from the center, or any vacation homes that cost more than £100 a night, we might exclude results that the user would consider to be a good compromise.

The `function_score` query gives (((("function*score query", "decay functions"))))(((("decay functions"))))us the ability to trade off one sliding scale (like location) against another sliding scale (like price), with a group of functions known as the _decay functions*.

The three decay functions--called `linear`, `exp`, and `gauss`—operate on numeric fields, date fields, or lat/lon geo-points. ((("linear function")))(((("exp (exponential) function")))(((("gauss (Gaussian) function")))) All three take the same parameters:

`origin` :: The *central point*, or the best possible value for the field. Documents that fall at the `origin` will get a full `_score` of `1.0`.

`scale` :: The rate of decay--how quickly the `_score` should drop the further from the `origin` that a document lies (for example, every £10 or every 100 meters).

`decay` :: The `_score` that a document at `scale` distance from the `origin` should receive. Defaults to `0.5`.

`offset` :: Setting a nonzero `offset` expands the central point to cover a range of values instead of just the single point specified by the `origin`. All values in the range `-offset <= origin <= +offset` will receive the full `_score` of `1.0`.

The only difference between these three functions is the shape of the decay curve. The difference is most easily illustrated with a graph (see <>).

[[img-decay-functions]] .Decay function curves image::images/elas_1705.png["The curves of the decay functions"]

The curves shown in <> all have their `origin`—the central point--set to `40`. The `offset` is `5`, meaning that all values in the range `40 - 5 <= value <= 40 + 5` are treated as though they were at the `origin`—they all get the full score of `1.0`.

Outside this range, the score starts to decay. The rate of decay is determined by the `scale` (which in this example is set to `5`), and the `decay` (which is set to the default of `0.5`). The result is that all three curves return a score of `0.5` at `origin +/- (offset + scale)`, or at points `30` and `50`.

The difference between `linear`, `exp`, and `gauss` is the shape of the curve at other points in the range:

- The `linear` funtion is just a straight line. Once the line hits zero, all values outside the line will return a score of `0.0`.
- The `exp` (exponential) function decays rapidly, then slows down.
- The `gauss` (Gaussian) function is bell-shaped--it decays slowly, then rapidly, then slows down again.

Which curve you choose depends entirely on how quickly you want the `_score` to decay, the further a value is from the `origin`.

To return to our example: our user would prefer to rent a vacation home close to the center of London (`{ "lat": 51.50, "lon": 0.12}`) and to pay no more than £100 a night, but our user considers price to be more important than distance. ((("gauss (Gaussian) function", "in function_score query"))) We could write this query as follows:

[source,json]

GET /_search { "query": { "function_score": { "functions": [{ "gauss": { "location": { <1> "origin": { "lat": 51.5, "lon": 0.12 }, "offset": "2km", "scale": "3km" } } }, { "gauss": { "price": { <2> "origin": "50", <3> "offset": "50", "scale": "20" } }, "weight": 2 <4> }] } }

}

<1> The `location` field is mapped as a `geo_point` .

<2> The `price` field is numeric.

<3> See <> for the reason that `origin` is `50` instead of `100` .

<4> The `price` clause has twice the weight of the `location` clause.

The `location` clause is(("location clause, Gaussian function example"))) easy to understand:

- We have specified an `origin` that corresponds to the center of London.
- Any location within `2km` of the `origin` receives the full score of `1.0` .
- Locations `5km` (`offset + scale`) from the centre receive a score of `0.5` .

[[Understanding-the-price-Clause]] === Understanding the price Clause

The `price` clause is a little trickier.(("price clause (Gaussian function example)")) The user's preferred price is anything up to £100, but this example sets the origin to £50. Prices can't be negative, but the lower they are, the better. Really, any price between £0 and £100 should be considered optimal.

If we were to set the `origin` to £100, then prices below £100 would receive a lower score. Instead, we set both the `origin` and the `offset` to £50. That way, the score decays only for any prices above £100 (`origin + offset`).

[TIP]

The `weight` parameter can be used to increase or decrease the contribution of individual clauses. (("weight parameter (in function_score query"))) The `weight` , which defaults to `1.0` , is multiplied by the score from each clause before the scores are combined with the specified `score_mode` .

==========

[[pluggable-similarites]] === Pluggable Similarity Algorithms

Before we move on from relevance and scoring, we will finish this chapter with a more advanced subject: pluggable similarity algorithms.(((("similarity algorithms", "pluggable")))((("relevance", "controlling", "using pluggable similarity algorithms"))) While Elasticsearch uses the <> as its default similarity algorithm, it supports other algorithms out of the box, which are listed in the http://bit.ly/14Eiw7f[Similarity Modules] documentation.

[[bm25]] ==== Okapi BM25

The most interesting competitor to TF/IDF and the vector space model is called http://en.wikipedia.org/wiki/Okapi_BM25[_Okapi BM25*], which is considered to be a _state-of-the-art* ranking function.((("BM25")))((("Okapi BM25", see="BM25"))) BM25 originates from the http://en.wikipedia.org/wiki/Probabilistic_relevance_model[probabilistic relevance model], rather than the vector space model, yet(("probabalistic relevance model")) the algorithm has a lot in common with Lucene's practical scoring function.

Both use of term frequency, inverse document frequency, and field-length normalization, but the definition of each of these factors is a little different. Rather than explaining the BM25 formula in detail, we will focus on the practical advantages that BM25 offers.

[[bm25-saturation]] ===== Term-frequency saturation

Both TF/IDF and BM25 use <> to distinguish between common (low value) words and uncommon (high value) words.(("inverse document frequency", "use by TF/IDF and BM25")) Both also recognize (see <>) that the more often a word appears in a document, the more likely is it that the document is relevant for that word.

However, common words occur commonly. (("BM25", "term frequency saturation")) The fact that a common word appears many times in one document is offset by the fact that the word appears many times in *all* documents.

However, TF/IDF was designed in an era when it was standard practice to remove the *most* common words (or *stopwords*, see <>) from the index altogether.(("stopwords", "removal from index")) The algorithm didn't need to worry about an upper limit for term frequency because the most frequent terms had already been removed.

In Elasticsearch, the `standard` analyzer--the default for `string` fields--doesn't remove stopwords because, even though they are words of little value, they do still have some value. The result is that, for very long documents, the sheer number of occurrences of words like `the` and `and` can artificially boost their weight.

BM25, on the other hand, does have an upper limit. Terms that appear 5 to 10 times in a document have a significantly larger impact on relevance than terms that appear just once or twice. However, as can be seen in <>, terms that appear 20 times in a document have almost the same impact as terms that appear a thousand times or more.

This is known as *nonlinear term-frequency saturation*.

[[img-bm25-saturation]] .Term frequency saturation for TF/IDF and BM25 image::images/elas_1706.png[Term frequency saturation for TF/IDF and BM25]

[[bm25-normalization]] ===== Field-length normalization

In <>, we said that Lucene considers shorter fields to have more weight than longer fields: the frequency of a term in a field is offset by the length of the field. However, the practical scoring function treats all fields in the same way. It will treat all `title` fields (because they are short) as more important than all `body` fields (because they are long).

BM25 also considers shorter fields to have more weight than longer fields, but it considers each field separately by taking the average length of the field into account. It can distinguish between a short `title` field and a `long` title field.

CAUTION: In <>, we said that the `title` field has a *natural* boost over the `body` field because of its length. This natural boost disappears with BM25 as differences in field length apply only within a single field.

[[bm25-tunability]] ===== Tuning BM25

One of the nice features of BM25 is that, unlike TF/IDF, it has two parameters that allow it to be tuned:

`k1` :: This parameter controls how quickly an increase in term frequency results in term-frequency saturation. The default value is `1.2`. Lower values result in quicker saturation, and higher values in slower saturation.

`b` :: This parameter controls how much effect field-length normalization should have. A value of `0.0` disables normalization completely, and a value of `1.0` normalizes fully. The default is `0.75`.

The practicalities of tuning BM25 are another matter. The default values for `k1` and `b` should be suitable for most document collections, but the optimal values really depend on the collection. Finding good values for your collection is a matter of adjusting, checking, and adjusting again.

[[relevance-conclusion]] === Relevance Tuning Is the Last 10%

In this chapter, we looked at a how Lucene generates scores based on TF/IDF. Understanding the score-generation process((("relevance", "controlling", "tuning relevance"))) is critical so you can tune, modulate, attenuate, and manipulate the score for your particular business domain.

In practice, simple combinations of queries will get you good search results. But to get *great* search results, you'll often have to start tinkering with the previously mentioned tuning methods.

Often, applying a boost on a strategic field or rearranging a query to emphasize a particular clause will be sufficient to make your results great. Sometimes you'll need more-invasive changes. This is usually the case if your scoring requirements diverge heavily from Lucene's word-based TF/IDF model (for example, you want to score based on time or distance).

With that said, relevancy tuning is a rabbit hole that you can easily fall into and never emerge. The concept of *most relevant* is a nebulous target to hit, and different people often have different ideas about document ranking. It is easy to get into a cycle of constant fiddling without any apparent progress.

We encourage you to avoid this (very tempting) behavior and instead properly instrument your search results. Monitor how often your users click the top result, the top 10, and the first page; how often they execute a secondary query without selecting a result first; how often they click a result and immediately go back to the search results, and so forth.

These are all indicators of how relevant your search results are to the user. If your query is returning highly relevant results, users will select one of the top-five results, find what they want, and leave. Irrelevant results cause users to click around and try new search queries.

Once you have instrumentation in place, tuning your query is simple. Make a change, monitor its effect on your users, and repeat as necessary. The tools outlined in this chapter are just that: tools. You have to use them appropriately to propel your search results into the *great* category, and the only way to do that is with strong measurement of user behavior.