



Elasticsearch

权威指南

clinton gormley , zachary tong 著

路小磊等 译



目錄

Introduction	0
入门	1
是什么	1.1
安装	1.2
API	1.3
文档	1.4
索引	1.5
搜索	1.6
聚合	1.7
小结	1.8
分布式	1.9
结语	1.10
分布式集群	2
空集群	2.1
集群健康	2.2
添加索引	2.3
故障转移	2.4
横向扩展	2.5
更多扩展	2.6
应对故障	2.7
数据	3
文档	3.1
索引	3.2
获取	3.3
存在	3.4
更新	3.5
创建	3.6
删除	3.7
版本控制	3.8
局部更新	3.9



Mget	3.10
批量	3.11
结语	3.12
分布式增删改查	4
路由	4.1
分片交互	4.2
新建、索引和删除	4.3
检索	4.4
局部更新	4.5
批量请求	4.6
批量格式	4.7
搜索	5
空搜索	5.1
多索引和多类型	5.2
分页	5.3
查询字符串	5.4
映射和分析	6
数据类型差异	6.1
确切值对决全文	6.2
倒排索引	6.3
分析	6.4
映射	6.5
复合类型	6.6
结构化查询	7
请求体查询	7.1
结构化查询	7.2
查询与过滤	7.3
重要的查询子句	7.4
过滤查询	7.5
验证查询	7.6
结语	7.7
排序	8
排序	8.1
字符串排序	8.2



相关性	8.3
字段数据	8.4
分布式搜索	9
查询阶段	9.1
取回阶段	9.2
搜索选项	9.3
扫描和滚屏	9.4
索引管理	10
创建删除	10.1
设置	10.2
配置分析器	10.3
自定义分析器	10.4
映射	10.5
根对象	10.6
元数据中的source字段	10.7
元数据中的all字段	10.8
元数据中的ID字段	10.9
动态映射	10.10
自定义动态映射	10.11
默认映射	10.12
重建索引	10.13
别名	10.14
深入分片	11
使文本可以被搜索	11.1
动态索引	11.2
近实时搜索	11.3
持久化变更	11.4
合并段	11.5
结构化搜索	12
查询准确值	12.1
组合过滤	12.2
查询多个准确值	12.3
包含，而不是相等	12.4



范围	12.5
处理 Null 值	12.6
缓存	12.7
过滤顺序	12.8
全文搜索	13
匹配查询	13.1
多词查询	13.2
组合查询	13.3
布尔匹配	13.4
增加子句	13.5
控制分析	13.6
关联失效	13.7
多字段搜索	14
多重查询字符串	14.1
单一查询字符串	14.2
最佳字段	14.3
最佳字段查询调优	14.4
多重匹配查询	14.5
最多字段查询	14.6
跨字段对象查询	14.7
以字段为中心查询	14.8
全字段查询	14.9
跨字段查询	14.10
精确查询	14.11
模糊匹配	15
Phrase matching	15.1
Slop	15.2
Multi value fields	15.3
Scoring	15.4
Relevance	15.5
Performance	15.6
Shingles	15.7
Partial_Matching	16
Postcodes	16.1



Prefix query	16.2
Wildcard Regexp	16.3
Match phrase prefix	16.4
Index time	16.5
Ngram intro	16.6
Search as you type	16.7
Compound words	16.8
Relevance	17
Scoring theory	17.1
Practical scoring	17.2
Query time boosting	17.3
Query scoring	17.4
Not quite not	17.5
Ignoring TFIDF	17.6
Function score query	17.7
Popularity	17.8
Boosting filtered subsets	17.9
Random scoring	17.10
Decay functions	17.11
Pluggable similarities	17.12
Conclusion	17.13
Language intro	18
Using	18.1
Configuring	18.2
Language pitfalls	18.3
One language per doc	18.4
One language per field	18.5
Mixed language fields	18.6
Conclusion	18.7
Identifying words	19
Standard analyzer	19.1
Standard tokenizer	19.2
ICU plugin	19.3



ICU tokenizer	19.4
Tidying text	19.5
Token normalization	20
Lowercasing	20.1
Removing diacritics	20.2
Unicode world	20.3
Case folding	20.4
Character folding	20.5
Sorting and collations	20.6
Stemming	21
Algorithmic stemmers	21.1
Dictionary stemmers	21.2
Hunspell stemmer	21.3
Choosing a stemmer	21.4
Controlling stemming	21.5
Stemming in situ	21.6
Stopwords	22
Intro	22.1
Using stopwords	22.2
Stopwords and performance	22.3
Divide and conquer	22.4
Phrase queries	22.5
Common grams	22.6
Relevance	22.7
Synonyms	23
Intro	23.1
Using synonyms	23.2
Synonym formats	23.3
Expand contract	23.4
Analysis chain	23.5
Multi word synonyms	23.6
Symbol synonyms	23.7
Fuzzy matching	24
Intro	24.1



Fuzziness	24.2
Fuzzy query	24.3
Fuzzy match query	24.4
Scoring fuzziness	24.5
Phonetic matching	24.6
Aggregations	25
overview	25.1
circuit breaker fd settings	25.2
filtering	25.3
facets	25.4
docvalues	25.5
eager	25.6
breadth vs depth	25.7
Conclusion	25.8
concepts buckets	25.9
basic example	25.10
add metric	25.11
nested bucket	25.12
extra metrics	25.13
bucket metric list	25.14
histogram	25.15
date histogram	25.16
scope	25.17
filtering	25.18
sorting ordering	25.19
approx intro	25.20
cardinality	25.21
percentiles	25.22
sigterms intro	25.23
sigterms	25.24
fielddata	25.25
analyzed vs not	25.26
地理坐标点	26



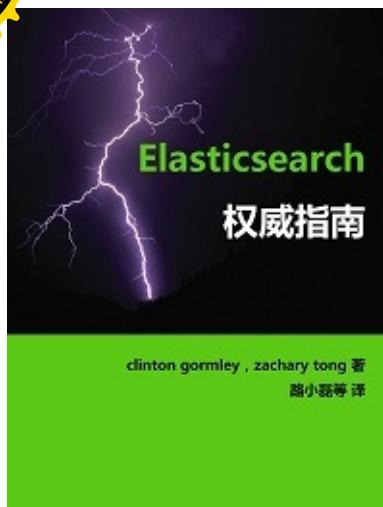
地理坐标点	26.1
通过地理坐标点过滤	26.2
地理坐标盒模型过滤器	26.3
地理距离过滤器	26.4
缓存地理位置过滤器	26.5
减少内存占用	26.6
按距离排序	26.7
Geohash	27
Geohash	27.1
Geohash映射	27.2
Geohash单元过滤器	27.3
地理位置聚合	28
地理位置聚合	28.1
按距离聚合	28.2
Geohash单元聚合器	28.3
范围（边界）聚合器	28.4
地理形状	29
地理形状	29.1
映射地理形状	29.2
索引地理形状	29.3
查询地理形状	29.4
在查询中使用已索引的形状	29.5
地理形状的过滤与缓存	29.6
关系	30
应用级别的Join操作	30.1
扁平化你的数据	30.2
Top hits	30.3
Concurrency	30.4
Concurrency solutions	30.5
嵌套	31
嵌套对象	31.1
嵌套映射	31.2
嵌套查询	31.3
嵌套排序	31.4



嵌套集合	31.5
Parent Child	32
Parent child	32.1
Indexing parent child	32.2
Has child	32.3
Has parent	32.4
Children agg	32.5
Grandparents	32.6
Practical considerations	32.7
Scaling	33
Shard	33.1
Overallocation	33.2
Kagillion shards	33.3
Capacity planning	33.4
Replica shards	33.5
Multiple indices	33.6
Index per timeframe	33.7
Index templates	33.8
Retiring data	33.9
Index per user	33.10
Shared index	33.11
Faking it	33.12
One big user	33.13
Scale is not infinite	33.14
Cluster Admin	34
Marvel	34.1
Health	34.2
Node stats	34.3
Other stats	34.4
Deployment	35
hardware	35.1
other	35.2
config	35.3



dont touch	35.4
heap	35.5
file descriptors	35.6
conclusion	35.7
cluster settings	35.8
Post Deployment	36
dynamic settings	36.1
logging	36.2
indexing perf	36.3
rolling restart	36.4
backup	36.5
restore	36.6
conclusion	36.7



Elasticsearch 权威指南（中文版）

阅读地址：[Elasticsearch权威指南（中文版）](#)

最新版阅读地址：[Elasticsearch: 权威指南](#)

感谢大家对**Elasticsearch**权威指南（中文版）的支持，现在新版的翻译工作已经迁移至
<https://github.com/elasticsearch-cn/elasticsearch-definitive-guide>

原书地址：[Elasticsearch the definitive guide](#)

原作者： clinton gormley , zachary tong

译者：[Looly](#)

参与翻译：

- [@iridiumcao](#)
- [@cvnx1](#)
- [@conan007ai](#)
- [@sailxjx](#)



- @wxlfight
- @xieyunzi
- @xdream86
- @conan007ai
- @williamzhao
- @dingusxp
- @birdroidcn
- @MaggieHwang

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/loolly/elasticsearch-definitive-guide-cn>

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

阅读地址：

<http://es-guide-preview.elasticsearch.cn/>

<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[@numbbbb](#)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

现在新版的翻译工作已经迁移至 <https://github.com/elasticsearch-cn/elasticsearch-definitive-guide>



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](http://elasticsearch.org/download) 下载最新版本的Elasticsearch。

```
curl -L -O http://download.elasticsearch.org/PATH/T0/VERSION.zip <1>
unzip elasticsearch-$VERSION.zip
cd elasticsearch-$VERSION
```

1. 从 [elasticsearch.org\download](http://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
```

启动后，如果只有本地可以访问，尝试修改配置文件 `elasticsearch.yml`

中 `network.host`(注意配置文件格式不是以 `#` 开头的要空一格，`:` 后要空一格)
为 `network.host: 0.0.0.0`

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>:<PORT>/<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
- PATH API路径（例如_count将返回集群中文档的数量），PATH可以包含多个组件，例如 _cluster/stats或者_nodes/stats/jvm
- 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 `` 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即可：**



```
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。所有的教程你既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。

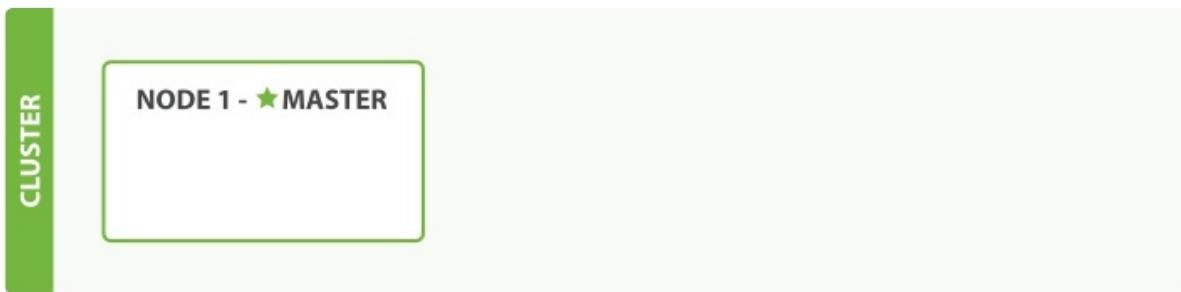


图1：只有一个空节点的集群

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，但是只有一个是最主要的：**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。

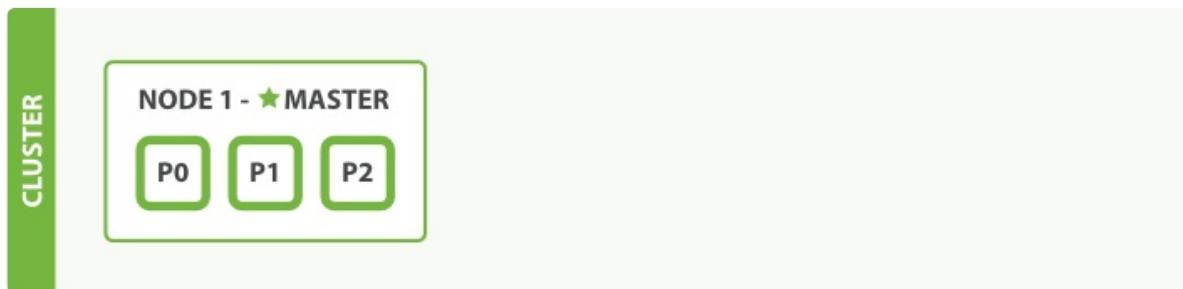
当索引创建完成的时候，主分片的数量就固定了，但是复制分片的数量可以随时调整。

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  
}
```

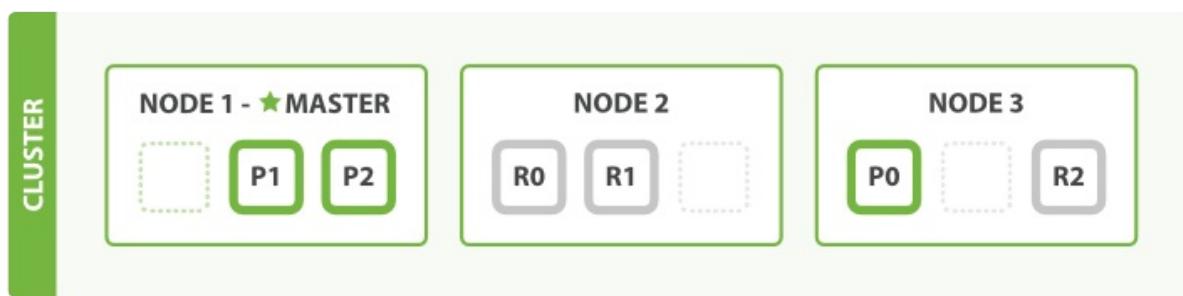
- <1> 集群的状态是 green .

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

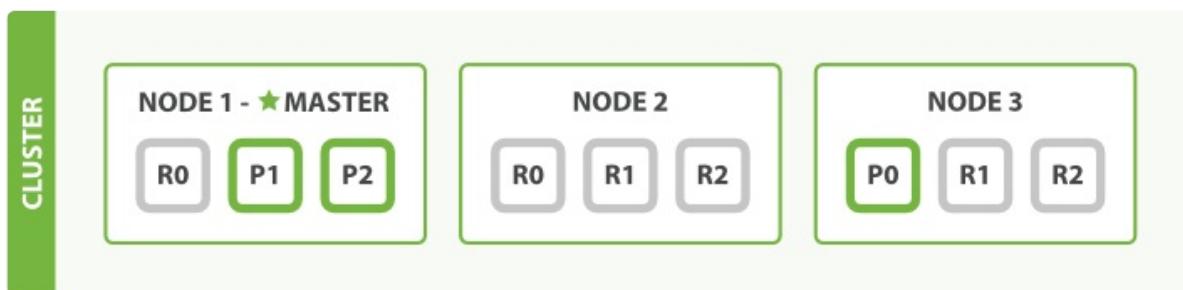
如果我们要扩展到6个以上的节点，要怎么做？

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：不是所有主分片都可用！

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。我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——“在这个类型下存储文档”。（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JSON
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即可，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。



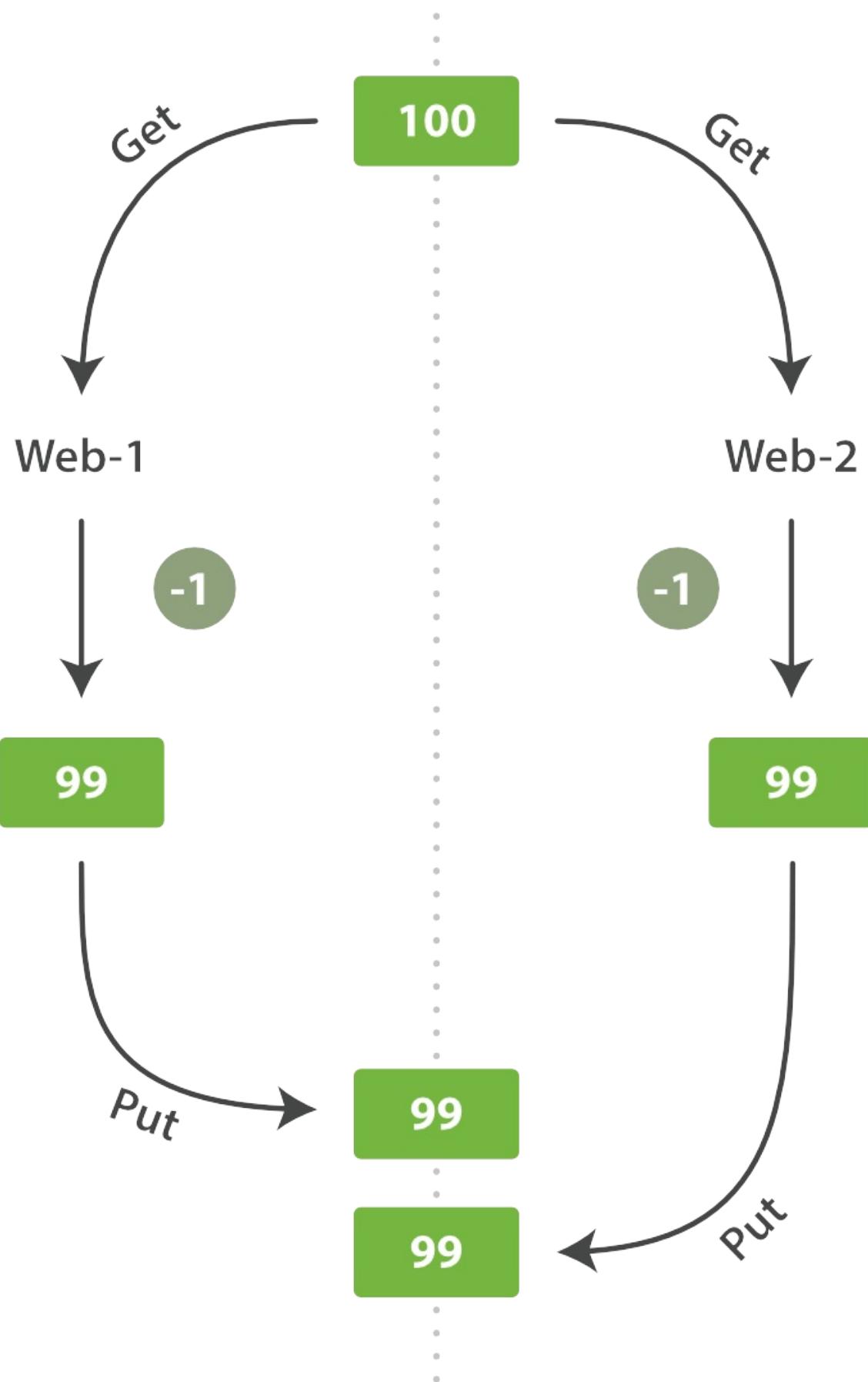
处理冲突

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/_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——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/_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/_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**)和重建索引(**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/_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**)来指定你要更新文档的版本。



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：

```
POST /_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：

```
POST /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：

```
POST /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。



更新时的批量操作

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...\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
<code>create</code>	当文档不存在时创建之。详见《创建文档》
<code>index</code>	创建新文档或替换已有文档。见《索引文档》和《更新文档》
<code>update</code>	局部更新文档。见《局部更新》
<code>delete</code>	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" }
```

如果未定义 `_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": 1 } } <2>  
{ "doc" : {"title" : "My updated blog post" } }
```

- <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，我们假设有一个三个节点的集群。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：



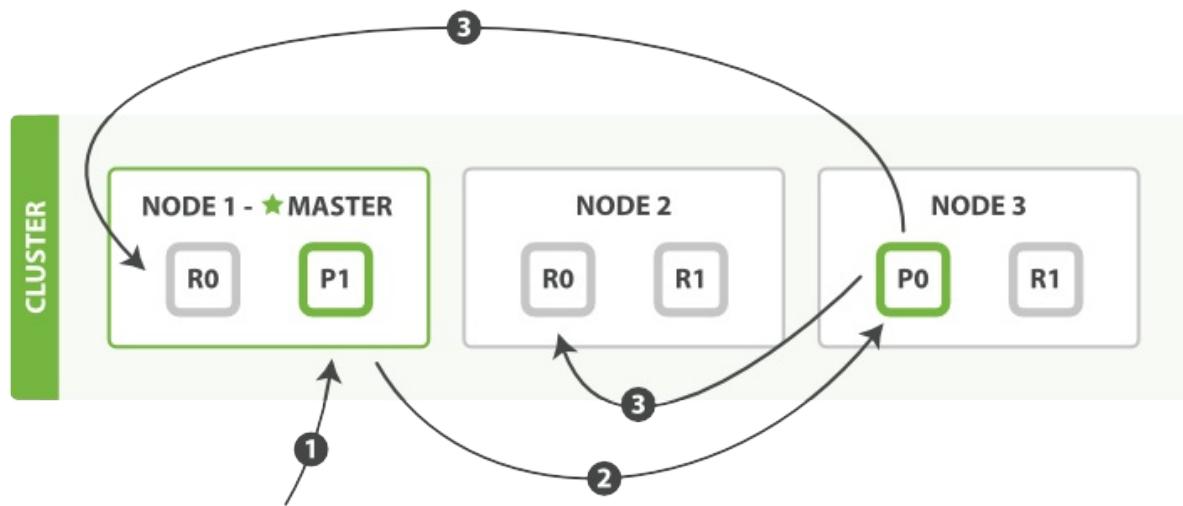
我们能够发送请求给集群中任意一个节点。每个节点都有能力处理任意请求。**每个节点都知道任意文档所在的节点，所以也可以将请求转发到需要的节点**。下面的例子中，我们将发送所有请求给 `Node 1`，这个节点我们将会称之为**请求节点(requesting node)**

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

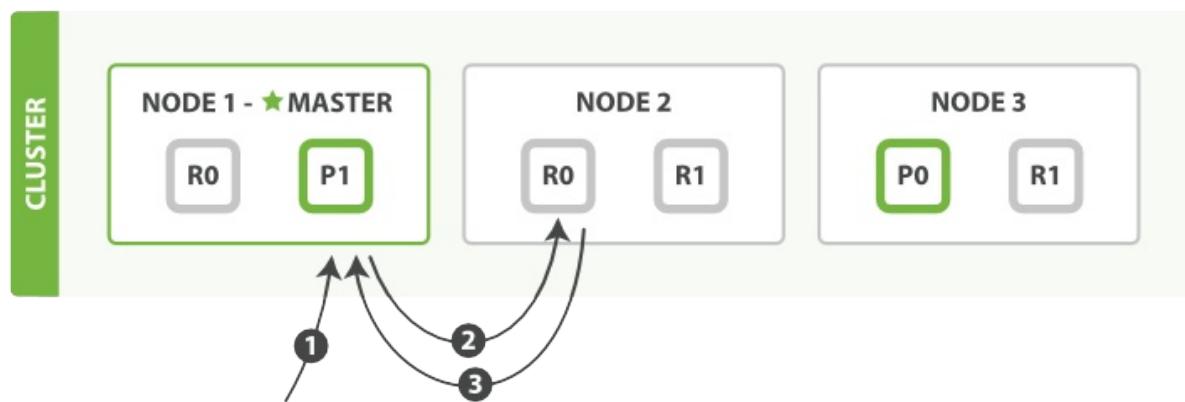
当分片副本不足时会怎样？Elasticsearch会等待更多的分片出现。**默认等待一分钟**。如果需要，你可以设置 `timeout` 参数让它终止的更早：`100` 表示100毫秒，`30s` 表示30秒。

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。为了避免这个问题，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：

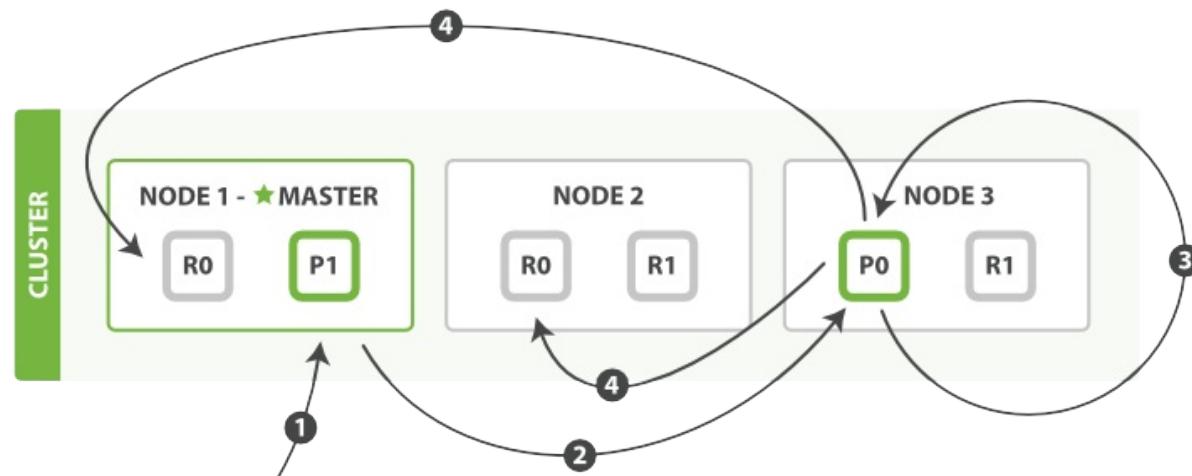
1. 客户端给 Node 1 发送get请求。
2. 节点使用文档的 `_id` 确定文档属于分片 `0`。分片 `0` 对应的复制分片在三个节点上都有。此时，它转发请求到 Node 2。
3. Node 2 返回文档(document)给 Node 1 然后返回给客户端。

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 `timeout` 参数。

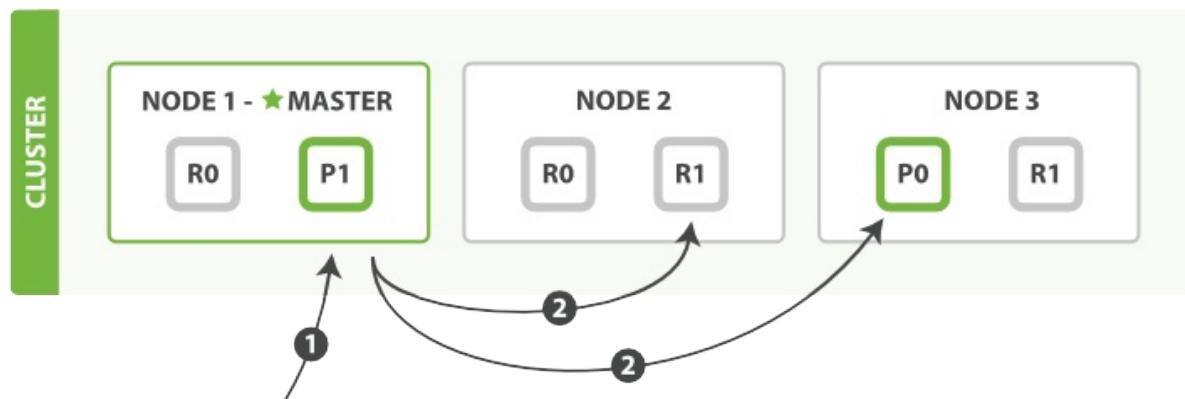
基于文档的复制

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。

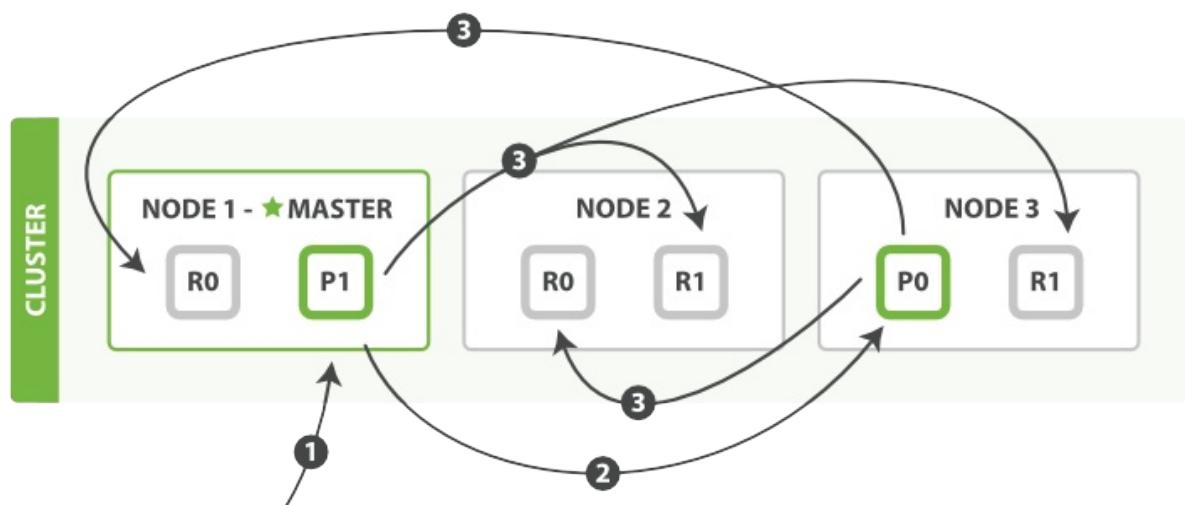
一旦接收到每个节点的应答，然后整理这些响应组合为一个单独的响应，最后返回给客户端。



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。一旦所有复制节点报告所有操作已成功完成，节点就报告 success 给请求节点，后者（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。但是只有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`：结果数，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%3Amy
```

"+" 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"
```

若没有指定字段，查询字符串搜索（即 `q=xxx`）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%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(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)，比如一篇推文(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		✓
The	✓	
brown	✓	✓
dog	✓	
dogs		✓
fox	✓	
foxes		✓
in		✓
jumped	✓	
lazy	✓	✓
leap		✓
over	✓	✓
quick	✓	
summer		✓
the	✓	

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	✓	✓
dog	✓	✓
fox	✓	✓
in		✓
jump	✓	✓
lazy	✓	✓
over	✓	✓
quick	✓	✓
summer		✓
the	✓	✓

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。

这个标记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，标记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**)，它们的工作是在标记化前处理字符串。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**)被标记化成独立的词。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）。

标记过滤

最后，每个词都通过所有标记过滤(**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，分词器和标记过滤器。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=Text to analyze
```

结果中每个节点代表一个词：



```
{  
  "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

为了能够把日期字段处理成日期，把数字字段处理成数字，把字符串字段处理成全文本（Full-text）或精确的字符串值，Elasticsearch需要知道每个字段里面都包含了什么类型。这些类型和字段的信息存储（包含）在映射（mapping）中。

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": "strict_date_optional_time||epoch_millis"
          },
          "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

虽然大多数情况下基本数据类型已经能够满足，但你也会经常需要自定义一些特殊类型(**fields**)，特别是字符串字段类型。自定义类型可以使你完成以下几点：



- 区分全文（full text）字符串字段和准确字符串字段（译者注：就是分词与不分词，全文的一般要分词，准确的就不需要分词，比如『中国』这个词。全文会分成『中』和『国』，但作为一个国家标识的时候我们是不需要分词的，所以它就应该是一个准确的字符串字段）。
- 使用特定语言的分析器（译者注：例如中文、英文、阿拉伯语，不同文字的断字、断词方式的差异）
- 优化部分匹配字段
- 指定自定义日期格式（译者注：这个比较好理解，例如英文的 Feb, 12, 2016 和中文的 2016年2月12日）
- 以及更多

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 analyzer。

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&text=Black-cats <1>
```

```
GET /gb/_analyze?field=tag&text=Black-cats <2>
```

<1> <2> 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串混合。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(bag of values)

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 ]
```

多层次对象

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hash、hashmap、dictionary 或者 associative array.



内部对象(**inner objects**)经常用于在另一个对象中嵌入一个实体或对象。例如，做为在 `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": { <3>  
                            "type": "object",  
                            "properties": {  
                                "full": { "type": "string" },  
                                "first": { "type": "string" },  
                                "last": { "type": "string" }  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```



<1> 根对象。

<2><3> 内部对象。

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` 并不了解内部对象。一个 `Lucene` 文件包含一个键-值对应的扁平表单。为了让 `Elasticsearch` 可以有效的索引内部对象，将文件转换为以下格式：

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

内部栏位可被归类至 `name`，例如 `"first"`。为了区别两个拥有相同名字的栏位，我们可以使用完整路径，例如 `"user.name.first"` 或甚至 类型 名称加上路径：`"tweet.user.name.first"`。

注意：在以上扁平化文件中，并没有栏位叫作 `user` 也没有栏位叫作 `user.name`。

`Lucene` 只索引阶层或简单的值，而不会索引复杂的资料结构。

对象-数组

内部对象数组

最后，一个包含内部对象的数组如何索引。我们有个数组如下所示：

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



此文件会如我们以上所说的被扁平化，但其结果会像如此：

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

{age: 35} 与 {name: Mary White} 之间的关联会消失，因每个多值的栏位会变成一个值集合，而非有序的阵列。这让我们可以知道：

- 是否有26岁的追随者？

但我们无法取得准确的资料如：

- 是否有26岁的追随者且名字叫**Alex Jones**？

关联内部对象可解决此类问题，我们称之为嵌套对象，我们之后会在嵌套对象中提到它。





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> 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：

- 叶子子句(*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，`must`，`must_not` 或者 `should`，如果可能的话：



```
{  
  "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。

以下实例查询的是邮件正文中含有“business opportunity”字样的星标邮件或收件箱中正文中含有“business opportunity”字样的非垃圾邮件：

```
{  
  "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，无论是叶子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：

- `created` 的日期范围是否在 `2013` 到 `2014` ?
- `status` 字段中是否包含单词 "published" ?
- `lat_lon` 字段中的地理位置与目标点相距是否不超过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。但是如果只有 `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

字符串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

译者注：多值字段是指同一个字段在ES索引中可以有多个含义，即可使用多个分析器(analyser)进行分词与排序，也可以不添加分析器，保留原值。

被分析器(analyser)处理过的字符称为 `analyzed field` (译者注：即已被分词并排序的字段，所有写入ES中的字段默认均会被`analyzed`)，`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 (译者注：未经分析器分词并排序的原字符串)。当然我们需要对字段进行全文本搜索的时候还必须使用被 `analyzed` 标记的字段。

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。而我们想要的是同一个字段中同时包含这两种索引方式，我们只需要改变索引(index)的mapping即可。方法是在所有核心字段类型上，使用通用参数 `fields` 对mapping进行修改)。比如，我们原有mapping如下：

```
"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 -- 等你需要的时候再回来回顾这一节就行。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` 字段中的出现频率。

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，值越小。

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*）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。

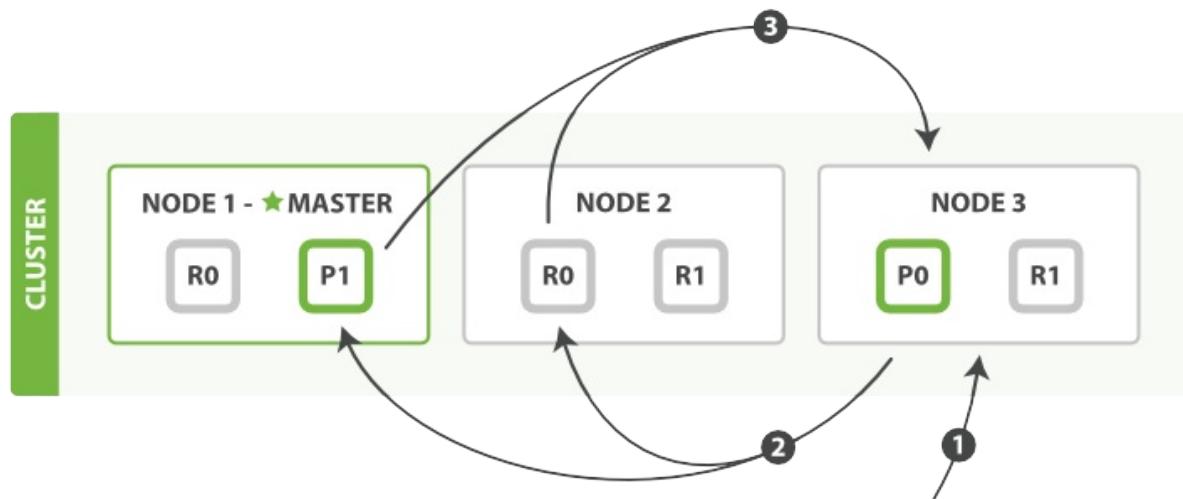


图1 分布式搜索查询阶段

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。

整个过程类似于归并排序算法，先分组排序再归并到一起，对于这种分布式场景非常适用。

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。

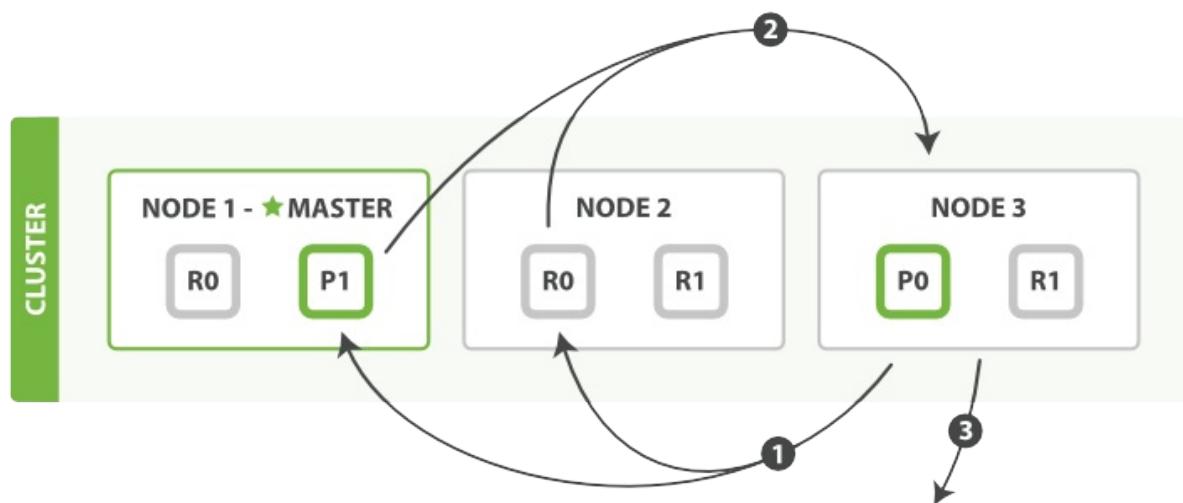


图2 分布式搜索取回阶段

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。她接受如下一些参数

`_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量结果。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)
c2Nhbjs10zEx0DpRNV9aY1VyUVM4U0NMd2pjWlJ3YWlB0zEx0TpRNV9aY1VyUVM4U0
NMD2pjWlJ3YWlB0zExNjpRNV9aY1VyUVM4U0NMd2pjWlJ3YWlB0zExNzpRNV9aY1V
yUVM4U0NMd2pjWlJ3YWlB0zEyMDpRNV9aY1VyUVM4U0NMd2pjWlJ3YWlB0zE7dG90Y
WxfaG10czox0w==
```

(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` 标记过滤器，被设计用来整理分词器触发的所有标记（但是目前什么都没做）。
- `lowercase` 标记过滤器，将所有标记转换为小写。
- `stop` 标记过滤器，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 E1 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，分词器和标记过滤器，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，标记过滤器）。

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 `Á` 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）或标记（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。

标记过滤器

分词结果的 标记流 会根据各自的情况，传递给特定的标记过滤器。

标记过滤器可能修改，添加或删除标记。我们已经提过 `lowercase` 和 `stop` 标记过滤器，但是 Elasticsearch 中有更多的选择。`stemmer` 标记过滤器将单词转化为他们的根形态（root form）。`ascii_folding` 标记过滤器会删除变音符号，比如从 `très` 转为 `tres`。`ngram` 和 `edge_ngram` 可以让标记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，分词器和标记过滤器：



```
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` 标记过滤器将词转为小写
3. 用 `stop` 标记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。一个索引只能包含一个类型(6.0以上)，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 处理一个位置的字段时，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": "text" },
          "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 `string` 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。

为了更高效的索引回索引中的文档，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，这意味着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中的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的文档列表多得多的信息。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: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。

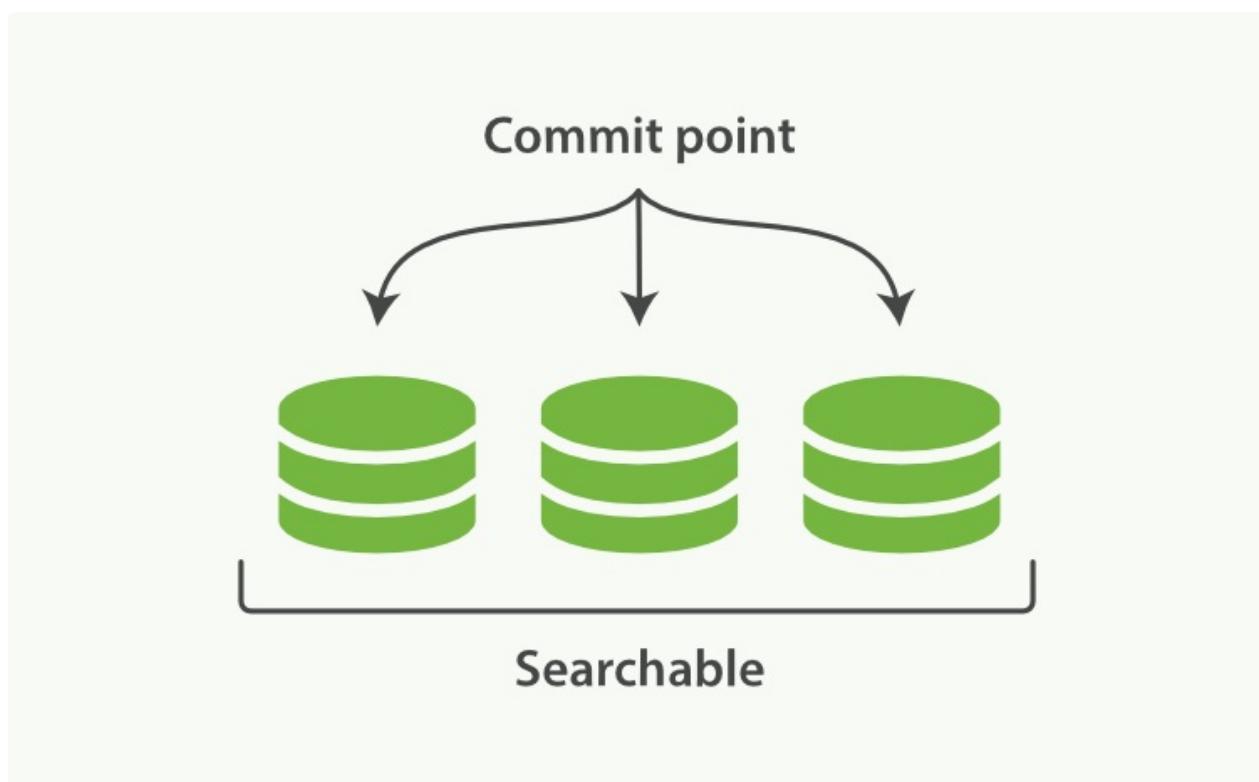
动态索引

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

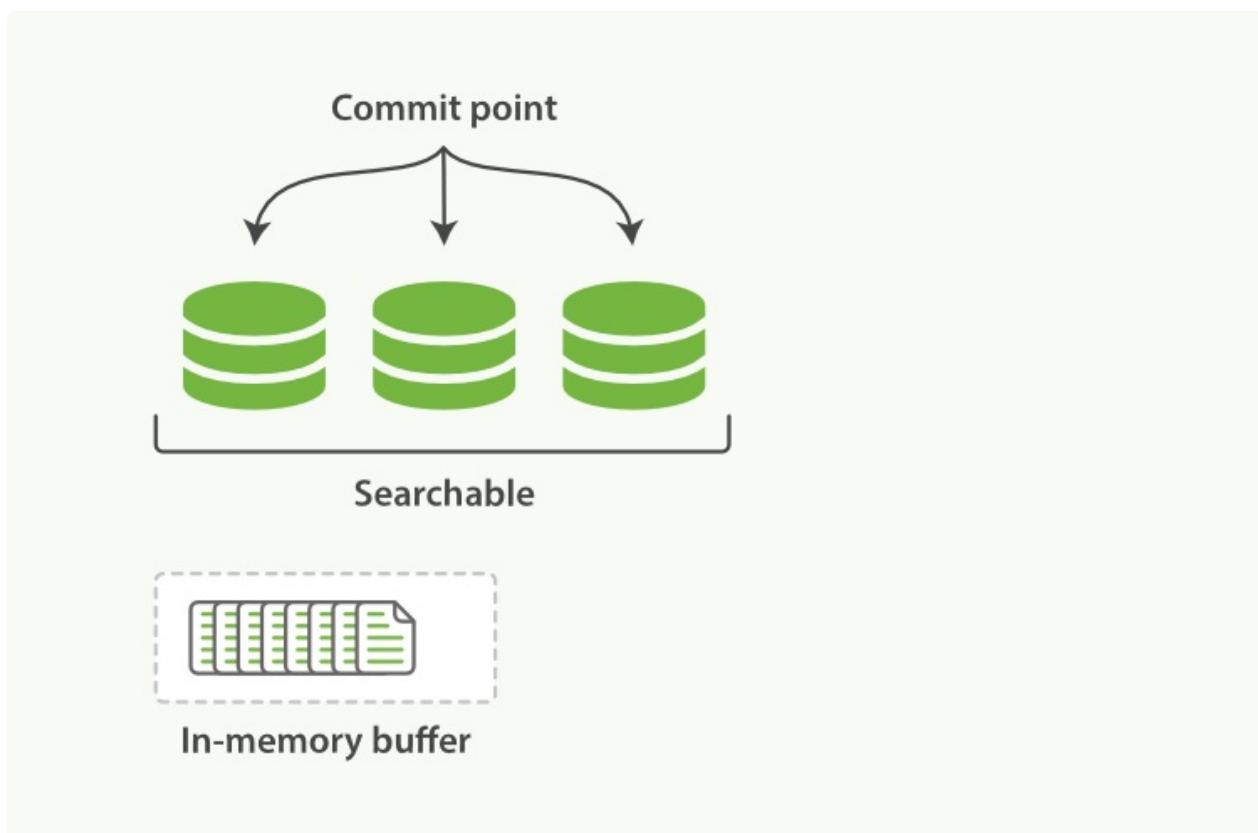
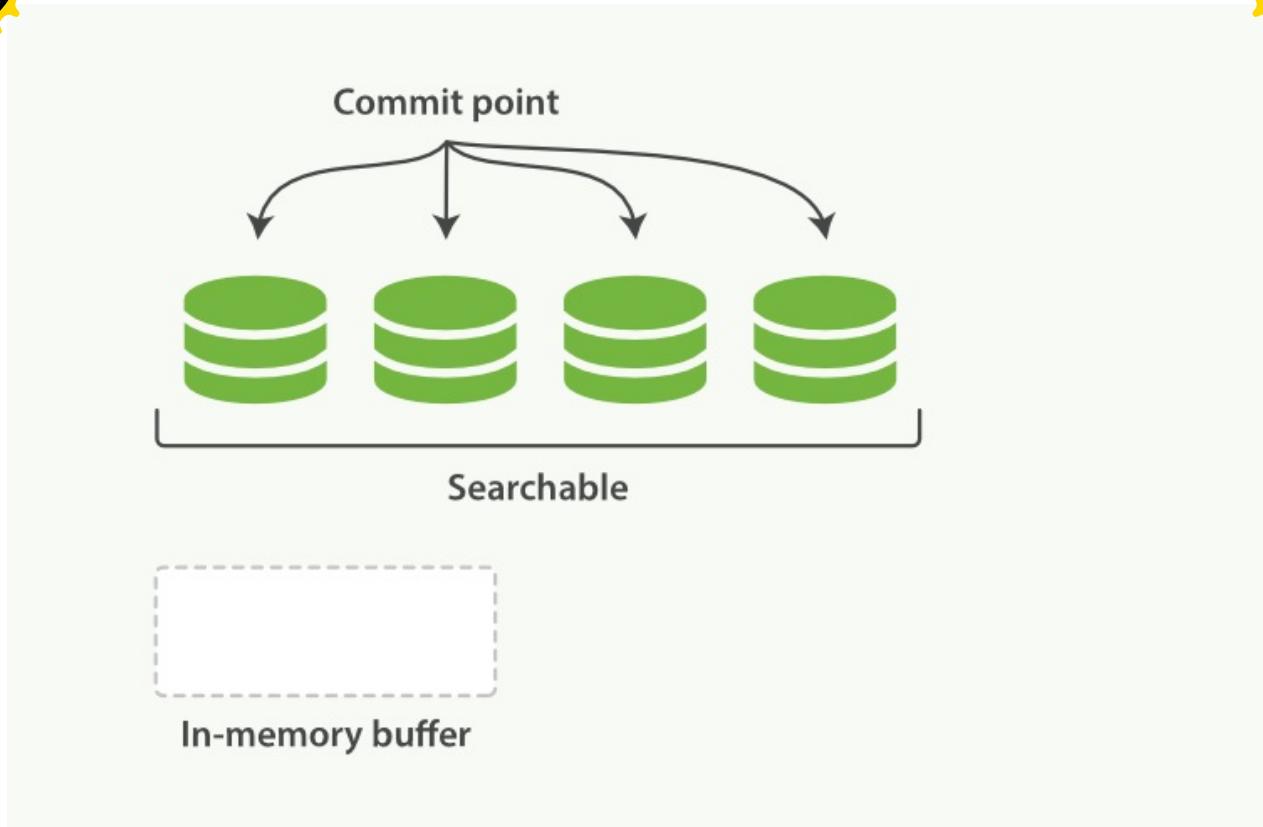


图3：提交后，新的段加到了提交点，缓存被清空



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

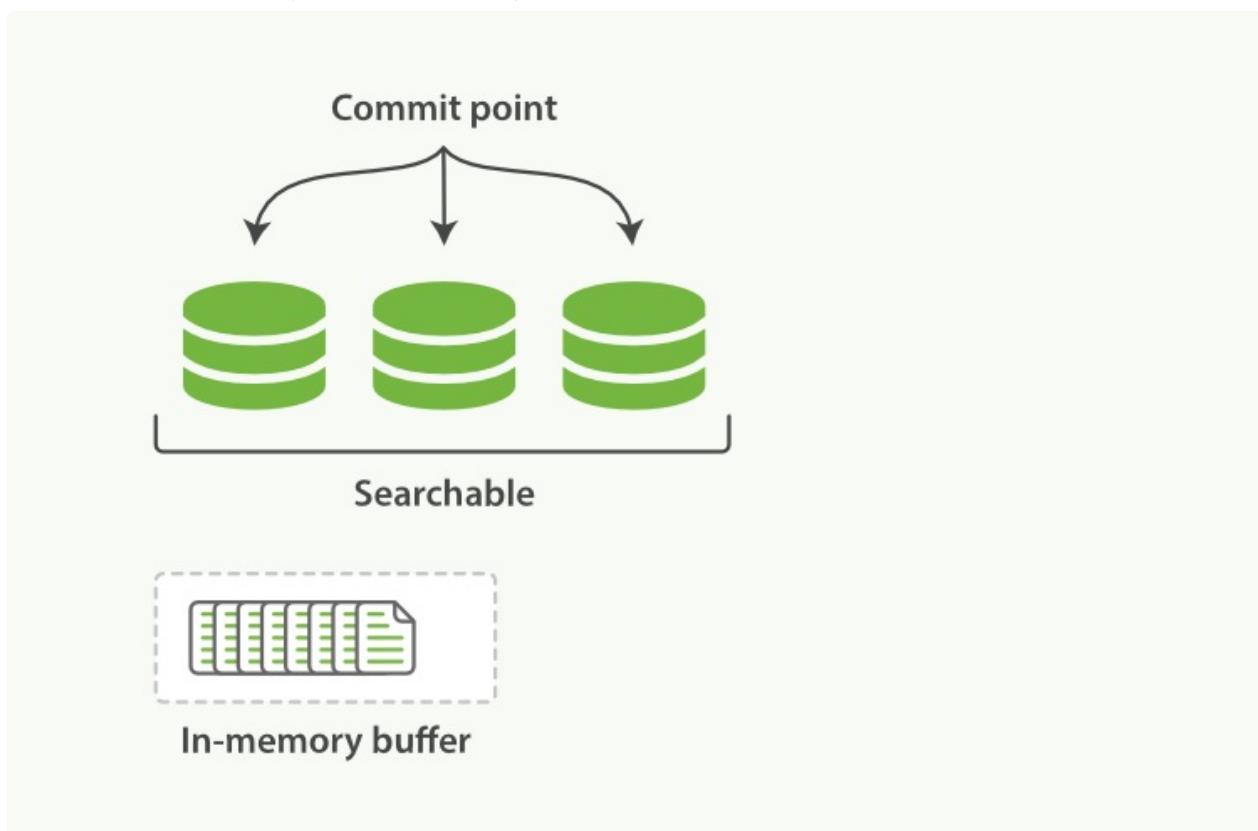
因为 `per-segment search` 机制，索引和搜索一个文档之间是有延迟的。新的文档会在几分钟内可以搜索，但是这依然不够快。

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，这意味着移除 `fsync`。

位于 Elasticsearch 和磁盘间的是文件系统缓存。如前所述，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。

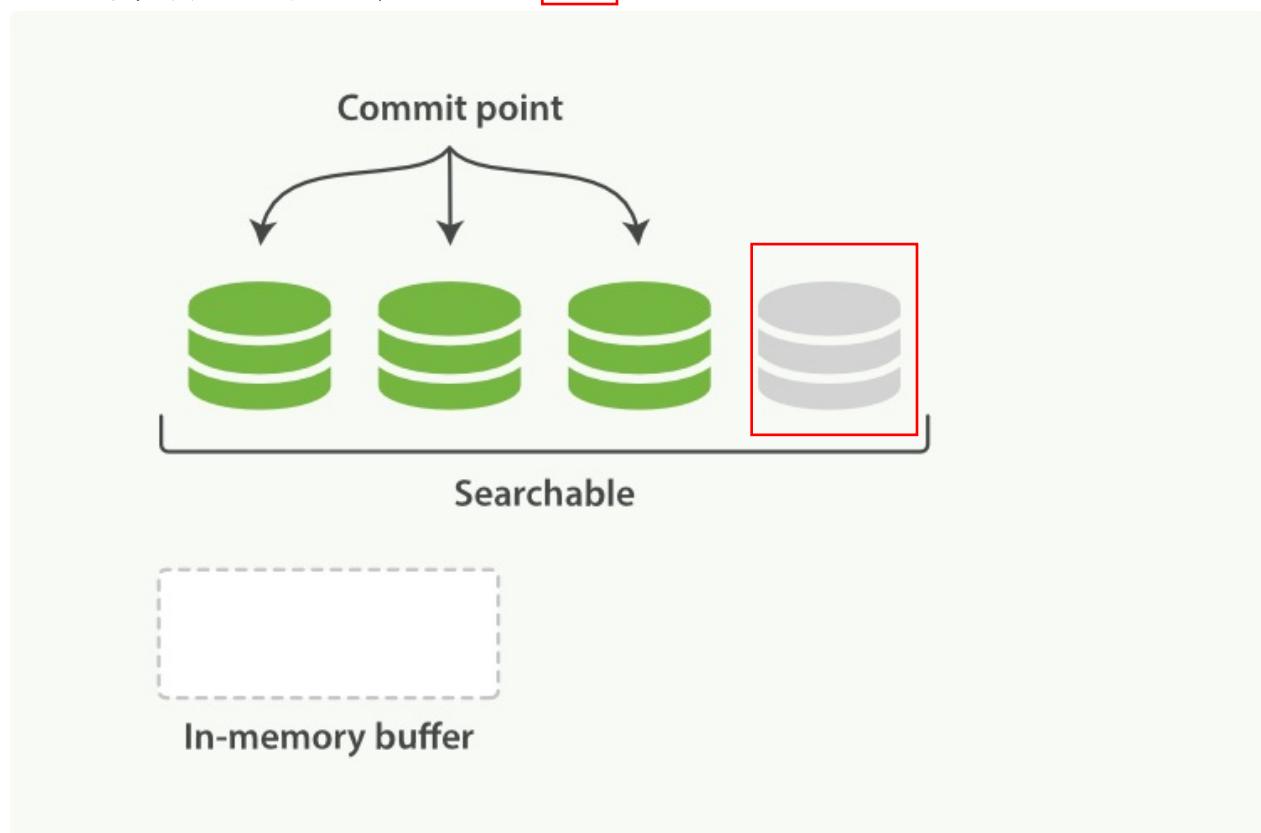
图1：内存缓存区有新文档的Lucene索引



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



refresh API

在Elasticsearch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。

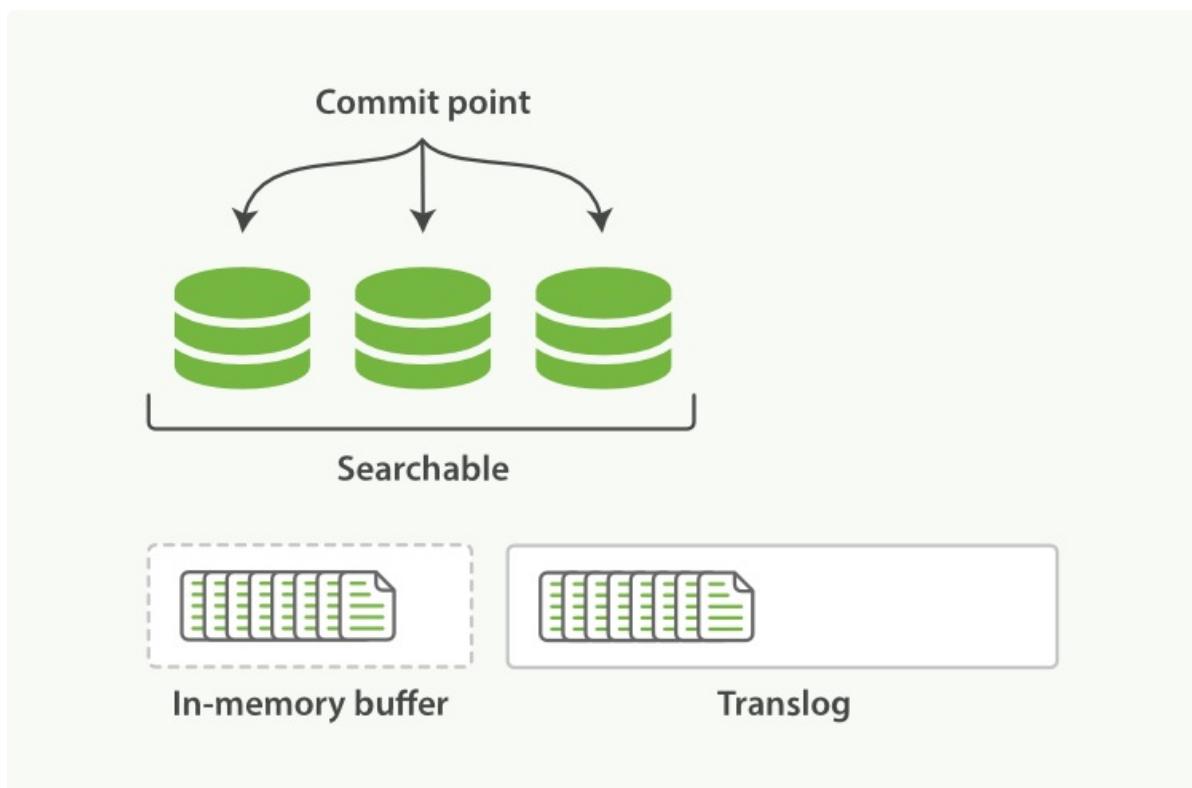
我们说过一次全提交同步段到磁盘，写提交点，这会列出所有的已知的段。在重启，或重新打开索引时，ES使用这次提交点决定哪些段属于当前的分片。

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：

- 当一个文档被索引，它被加入到内存缓冲，同时加到事务日志。

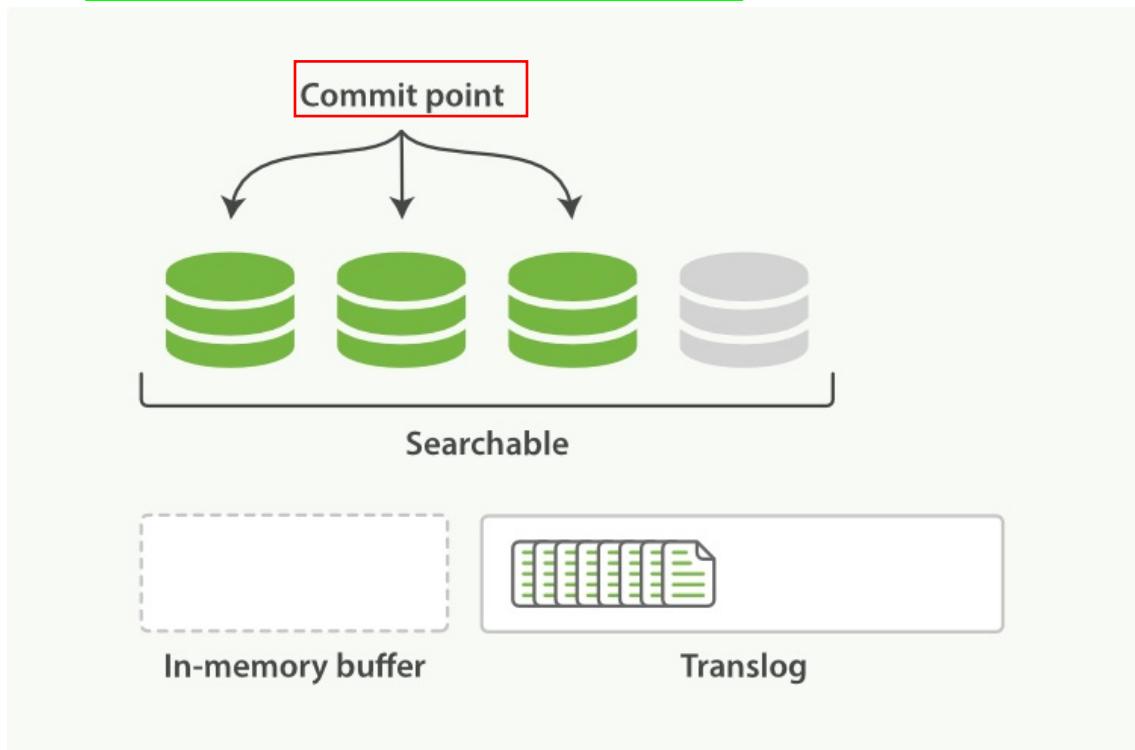
图1：新的文档加入到内存缓存，同时写入事务日志



- refresh使得分片的进入如下图描述的状态。每秒分片都进行refeash：

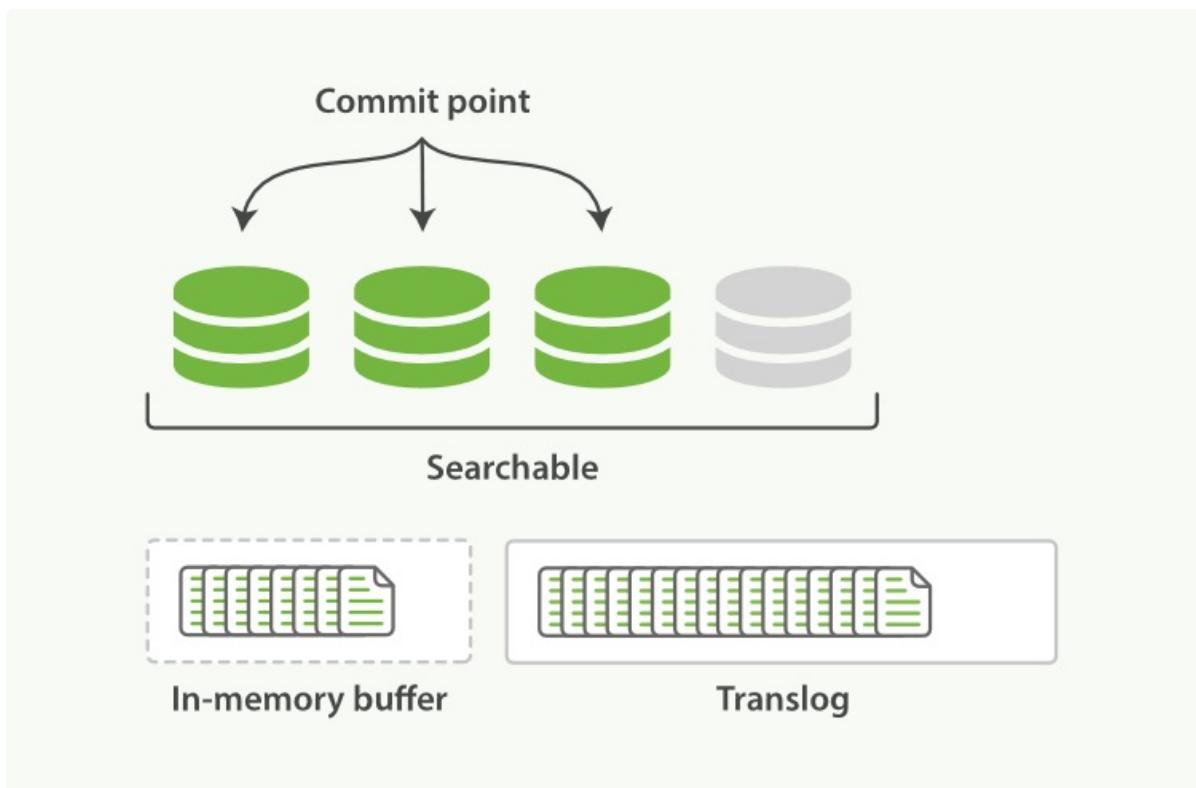
- 内存缓冲区的文档写入到段中，但没有`fsync`。
- 段被打开，使得新的文档可以搜索。
- 缓存被清除

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：

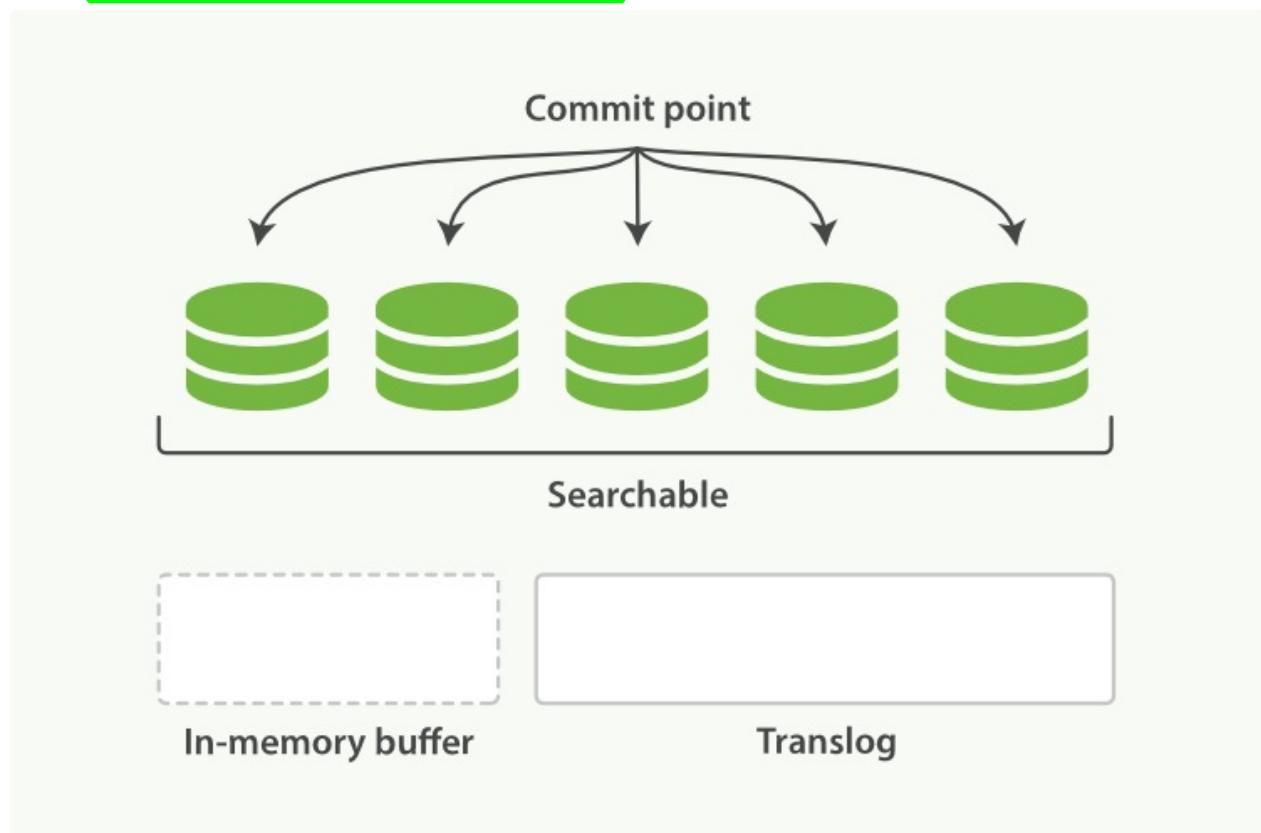
- 内存缓存区的所有文档会写入到新段中。
- 清除缓存

- 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。当你尝试用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` 可以用来进行一次手动`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。**

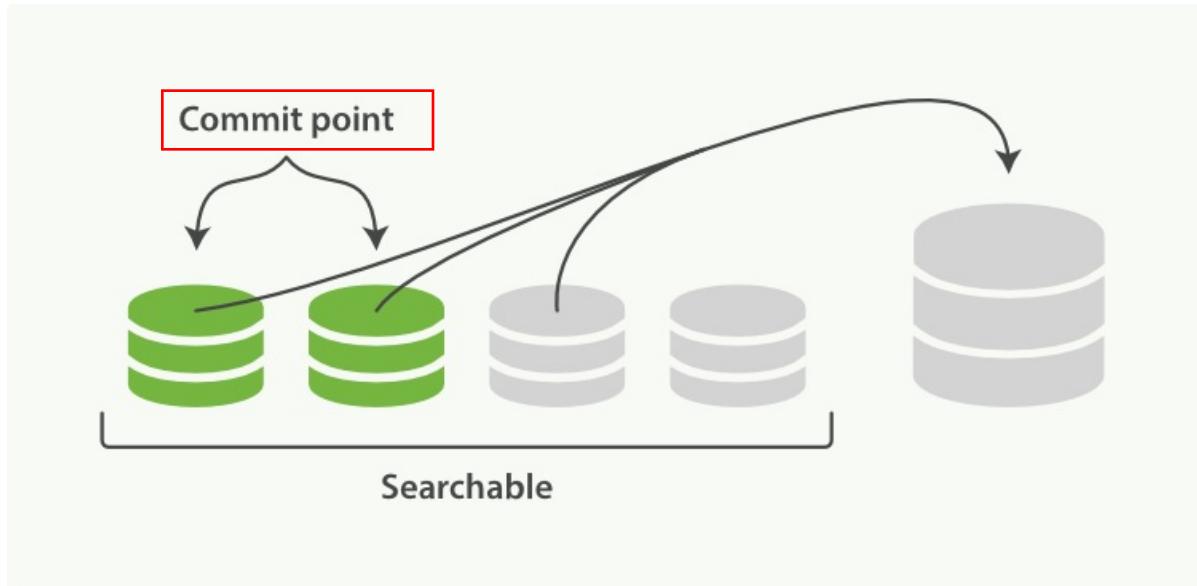
ES通过后台合并段解决这个问题。小段被合并成大段，再合并成更大的段。

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。

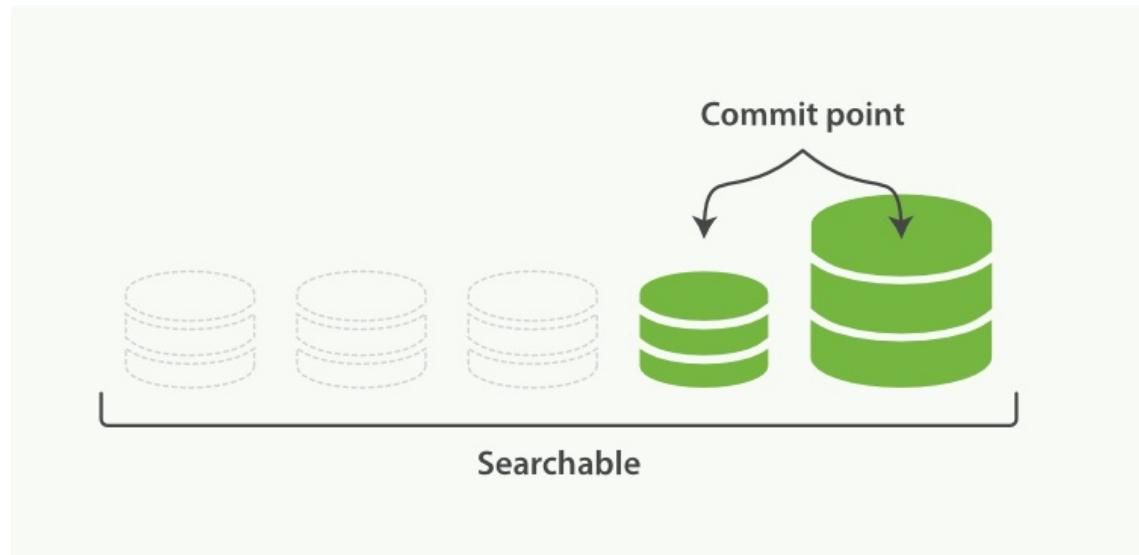
图1：两个提交的段和一个未提交的段合并为了一个更大的段



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

`optimize 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 `optimize 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，`optimize 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内——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 `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" : "xhdः",  
    "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标记，而不是一个完整的标记来表示 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，因为这个标记不在我们的倒排索引中。相反，那里有上面列出的四个标记。

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> 虽然这个文章除了 `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
<code>open_source</code>	2
<code>search</code>	1, 2

你怎么可能储存一个在数据结构不存在的字段呢？倒排索引是标记和包含它们的文档的一个简单列表。假如一个字段不存在，它就没有任何标记，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"  
}
```

所以我们要使用 `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。



全文检索

我们已经介绍了简单的结构化查询，下面开始介绍全文检索：怎样对全文字段(full-text fields)进行检索以找到相关度最高的文档。

全文检索最重要的两个方面是：

- 相关度(Relevance)

根据文档与查询的相关程度对结果集进行排序的能力。**相关度可以使用TF/IDF、地理位置相近程度、模糊相似度或其他算法计算。**

- 分析(Analysis)

将一段文本转换为一组唯一的、标准化了的标记(token)，用以(a)创建倒排索引，(b)查询倒排索引。

注意，**一旦提到相关度和分析，指的都是查询(queries)而非过滤器(filters)。**

基于短语 vs. 全文

虽然**所有的查询都会进行相关度计算，但不是所有的查询都有分析阶段**。而且像 `bool` 或 `function_score` 这样的查询并不在文本字段执行。文本查询可以分为两大类：

1. 基于短语(Term-based)的查询：

像 `term` 或 `fuzzy` 一类的查询是低级查询，它们没有分析阶段。这些查询在单一的短语上执行。例如对单词 '`Foo`' 的 `term` 查询会在倒排索引里**精确地查找 '`Foo`' 这个词，并对每个包含这个单词的文档计算TF/IDF相关度 '`_score`'**。

牢记 `term` 查询只在倒排查询里**精确地查找特定短语**，而不会匹配短语的其它变形，如 `foo` 或 `FOO`。不管短语怎样被加入索引，都只匹配倒排索引里的准确值。如果你在一个设置了 '`not_analyzed`' 的字段为 `'["Foo", "Bar"]'` 建索引，或者在一个用 '`whitespace`' 解析器解析的字段为 '`Foo Bar`' 建索引，都会在倒排索引里加入两个索引 '`Foo`' 和 '`Bar`'。

2. 全文(Full-text)检索

`match` 和 `query_string` 这样的查询是高级查询，它们会对字段进行分析：

- 如果检索一个 '`date`' 或 '`integer`' 字段，它们会把查询语句作为日期或者整数格式数据。



- 如果检索一个准确值('not_analyzed')字符串字段，它们会把整个查询语句作为一个短语。
- 如果检索一个全文('analyzed')字段，查询会先用适当的解析器解析查询语句，产生需要查询的短语列表。然后对列表中的每个短语执行低级查询，合并查询结果，得到最终的文档相关度。我们将会在后续章节讨论这一过程的细节。

我们很少需要直接使用基于短语的查询。**通常我们会想要检索全文，而不是单独的短语，使用高级的全文检索会更简单（全文检索内部最终还是使用基于短语的查询）。**

[提示]

如果确实要查询一个准确值字段('not_analyzed')，需要考虑使用查询还是过滤器。

单一短语的查询通常相当于是/否问题，用过滤器可以更好的描述这类查询，并且过滤器缓存可以提升性能：

```
GET /_search
{
  "query": {
    "filtered": {
      "filter": {
        "term": { "gender": "female" }
      }
    }
  }
}
```



匹配查询

不管你搜索什么内容，`match` 查询是你首先需要接触的查询。它是一个高级查询，意味着`match` 查询知道如何更好的处理全文检索和准确值检索。

这也就是说，`match` 查询的一个主要用途是进行全文搜索。让我们通过一个小例子来看一下全文搜索是如何工作的。

索引一些数据

首先，我们使用`bulk` API来创建和索引一些文档：

```
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> 删除已经存在的索引(如果索引存在)

<2> 然后，`关联失效`这一节解释了为什么我们创建该索引的时候只使用一个主分片。

单词查询

第一个例子解释了当使用`match` 查询进行单词全文搜索时发生了什么：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": "QUICK!"
    }
  }
}
```



// SENSE: 100_Full_Text_Search/05_Match_query.json

Elasticsearch通过下面的步骤执行 `match` 查询：

1. 检查`field`类型

`title` 字段是一个字符串(`analyzed`)，所以该查询字符串也需要被分析(`analyzed`)

2. 分析查询字符串

查询词 `QUICK!` 经过标准分析器的分析后变成单词 `quick`。因为我们只有一个查询词，因此 `match` 查询可以以一种低级别 `term` 查询的方式执行。

3. 找到匹配的文档

`term` 查询在倒排索引中搜索 `quick`，并且返回包含该词的文档。在这个例子中，返回的文档是1, 2, 3。

4. 为每个文档打分

`term` 查询综合考虑词频（每篇文档 `title` 字段包含 `quick` 的次数）、逆文档频率（在全部文档中 `title` 字段包含 `quick` 的次数）、包含 `quick` 的字段长度（长度越短越相关）来计算每篇文档的相关性得分 `_score`。（更多请见相关性介绍）

这个过程之后我们将得到以下结果（简化后）：

```
"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> 文档1最相关，因为 `title` 最短，意味着 `quick` 在语义中起比较大的作用。

<2> 文档3比文档2更相关，因为在文档3中 `quick` 出现了两次。





多词查询

如果一次只能查询一个关键词，全文检索将会很不方便。幸运的是，用 `match` 查询进行多词查询也很简单：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": "BROWN DOG!"
    }
  }
}
```

上面这个查询返回以下结果集：



```
{  
    "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> 文档4的相关度最高，因为包含两个"brown"和一个"dog"。

<2> 文档2和3都包含一个"brown"和一个"dog"，且'title'字段长度相同，所以相关度相等。

<3> 文档1只包含一个"brown"，不包含"dog"，所以相关度最低。

因为 `match` 查询需要查询两个关键词： "brown" 和 "dog" ，在内部会执行两个 `term` 查询并综合二者的结果得到最终的结果。`match` 的实现方式是将两个 `term` 查询放入一个 `bool` 查询， `bool` 查询在之前的章节已经介绍过。

重要的一点是， 'title' 字段包含至少一个查询关键字的文档都被认为是符合查询条件的。匹配的单词数越多，文档的相关度越高。

提高精度



匹配包含任意个数查询关键字的文档可能会得到一些看似不相关的结果，这是一种霰弹策略 (shotgun approach)。然而我们可能想得到包含所有查询关键字的文档。换句话说，我们想得到的是匹配 'brown AND dog' 的文档，而非 'brown OR dog'。

`match` 查询接受一个 'operator' 参数，默认值为 `or`。如果要求所有查询关键字都匹配，可以更改参数值为 `and`：

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": { <1>
        "query": "BROWN DOG!",
        "operator": "and"
      }
    }
  }
}
```

<1> 为了加入``operator``参数，`match`查询的结构有一些不同。

这个查询会排除文档1，因为文档1只包含了一个查询关键词。

控制精度

在 `all` 和 `any` 之间的选择有点过于非黑即白。如果用户指定了5个查询关键字，而一个文档只包含了其中的4个？将 'operator' 设置为 `'and'` 会排除这个文档。

有时这的确是用户想要的结果。**但在大多数全文检索的使用场景下，用户想得到相关的文档，排除那些不太可能相关的文档。**换句话说，我们需要介于二者之间的选项。

`match` 查询有 `'minimum_should_match'` 参数，参数值表示被视为相关的文档必须匹配的关键字个数。参数值可以设为整数，也可以设置为百分数。因为不能提前确定用户输入的查询关键词个数，使用百分数也很合理。

```
GET /my_index/my_type/_search
{
  "query": {
    "match": {
      "title": {
        "query": "quick brown dog",
        "minimum_should_match": "75%"
      }
    }
  }
}
```



当 `'minimum_should_match'` 被设置为百分数时，查询进行如下：在上面的例子里，`'75%'` 会被下舍为 `'66.6%'`，也就是2个关键词。不论参数值为多少，进入结果集的文档至少应匹配一个关键词。

[提示]

`'minimum_should_match'` 参数很灵活，根据用户输入的关键词个数，可以采用不同的匹配规则。更详细的内容可以查看[文档](#)。

要全面理解 `match` 查询是怎样处理多词查询，我们需要了解怎样用 `bool` 查询合并多个查询。



组合查询

在《组合过滤》中我们讨论了怎样用布尔过滤器组合多个用 `and` , `or` , `and not` 逻辑组成的过滤子句，在查询中，布尔查询充当着相似的作用，但是有一个重要的区别。

过滤器会做一个判断：是否应该将文档添加到结果集？然而查询会做更精细的判断。他们不仅决定一个文档是否要添加到结果集，而且还要计算文档的相关性（*relevant*）。

像过滤器一样，布尔查询接受多个用 `must` , `must_not` , `and` `should` 的查询子句，例：

```
GET /my_index/my_type/_search
{
  "query": {
    "bool": {
      "must": { "match": { "title": "quick" }},
      "must_not": { "match": { "title": "lazy" }},
      "should": [
        { "match": { "title": "brown" }},
        { "match": { "title": "dog" }}
      ]
    }
  }
}
```

在前面的查询中，凡是满足 `title` 字段中包含 `quick`，但是不包含 `lazy` 的文档都会在查询结果中。到目前为止，布尔查询的作用非常类似于布尔过滤的作用。

当 `should` 过滤器中有两个子句时不同的地方就体现出来了，下面例子就可以体现：一个文档不需要同时包含 `brown` 和 `dog`，但如果同时有这两个词，这个文档的相关性就更高。



```
{  
  "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> 文档3的得分更高，是因为它同时包含了 brown 和 dog 。

得分计算

布尔查询通过把所有符合 must 和 should 的子句得分加起来，然后除以 must 和 should 子句的总数为每个文档计算相关性得分。

must_not 子句并不影响得分；他们存在的意义是排除已经被包含的文档。

精度控制

所有的 must 子句必须匹配，并且所有的 must_not 子句必须不匹配，但是多少 should 子句应该匹配呢？默认的，不需要匹配任何 should 子句，一种情况例外：如果没有 must 子句，就必须至少匹配一个 should 子句。

像我们控制 match 查询的精度一样，我们也可以通过 minimum_should_match 参数控制多少 should 子句需要被匹配，这个参数可以是正整数，也可以是百分比。



```
GET /my_index/my_type/_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "brown" } },
        { "match": { "title": "fox" } },
        { "match": { "title": "dog" } }
      ],
      "minimum_should_match": 2 <1>
    }
  }
}
```

<1> 这也可以用百分比表示

结果集仅包含 title 字段中有 "brown" 和 "fox"， "brown" 和 "dog"，或 "fox" 和 "dog" 的文档。如果一个文档包含上述三个条件，那么它的相关性就会比其他仅包含三者中的两个条件的文档要高。



match 匹配怎么当成布尔查询来使用

到现在为止，你可能已经意识到在一个布尔查询中多字段 `match` 查询仅仅包裹了已经生成的 `term` 查询。通过默认的 `or` 操作符，每个 `term` 查询都会像一个 `should` 子句一样被添加，只要有一个子句匹配就可以了。下面的两个查询是等价的：

```
{  
  "match": { "title": "brown fox"}  
}
```

```
{  
  "bool": {  
    "should": [  
      { "term": { "title": "brown" }},  
      { "term": { "title": "fox" }}  
    ]  
  }  
}
```

通过 `and` 操作符，所有的 `term` 查询会像 `must` 子句一样被添加，因此所有的子句都必须匹配。下面的两个查询是等价的：

```
{  
  "match": {  
    "title": {  
      "query": "brown fox",  
      "operator": "and"  
    }  
  }  
}
```

```
{  
  "bool": {  
    "must": [  
      { "term": { "title": "brown" }},  
      { "term": { "title": "fox" }}  
    ]  
  }  
}
```

如果 `minimum_should_match` 参数被指定，`match` 查询就直接被转换成一个 `bool` 查询，下面两个查询是等价的：



```
{  
    "match": {  
        "title": {  
            "query": "quick brown fox",  
            "minimum_should_match": "75%"  
        }  
    }  
}
```

```
{  
    "bool": {  
        "should": [  
            { "term": { "title": "brown" }},  
            { "term": { "title": "fox" }},  
            { "term": { "title": "quick" }}  
        ],  
        "minimum_should_match": 2 <1>  
    }  
}
```

<1>因为只有三个子句，所以 `minimum_should_match` 参数在 `match` 查询中的值 `75%` 就下舍到了 `2`。3个 `should` 子句中至少有两个子句匹配。

当然，我们通常写这些查询类型的时候还是使用 `match` 查询的，但是理解 `match` 查询在内部是怎么工作的可以让你在任何你需要使用的时候更加得心应手。**有些情况仅仅使用一个 `match` 查询是不够的，比如给某些查询词更高的权重**。这种情况我们会在下一节看个例子。



提高查询得分

当然，`bool` 查询不仅仅是组合多个简单的一个词的 `match` 查询。他可以组合任何其他查询，包括 `bool` 查询。`bool` 查询通常会通过组合几个不同查询的得分为每个文档调整相关性得分。

假设我们想查找关于“full-text search”的文档，但是我们又想给涉及到“Elasticsearch”或者“Lucene”的文档更高的权重。我们的用意是想涉及到“Elasticsearch”或者“Lucene”的文档的相关性得分会比那些没有涉及到的文档的得分要高，也就是说这些文档会出现在结果集更靠前的位置。

一个简单的 `bool` 查询允许我们写出像下面一样的非常复杂的逻辑：

```
GET /_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "content": { (1)
            "query": "full text search",
            "operator": "and"
          }
        }
      },
      "should": [ (2)
        { "match": { "content": "Elasticsearch" }},
        { "match": { "content": "Lucene" } }
      ]
    }
  }
}
```

1. `content` 字段必须包含 `full`, `text`, `search` 这三个单词。
2. 如果 `content` 字段也包含了“Elasticsearch”或者“Lucene”，则文档会有一个更高的得分。

匹配的 `should` 子句越多，文档的相关性就越强。到目前为止一切都很好。但是如果我想给包含“Lucene”一词的文档比较高的得分，甚至给包含“Elasticsearch”一词更高的得分要怎么做呢？

我们可以在任何查询子句中指定一个 `boost` 值来控制相对权重，默认值为1。一个大于1的 `boost` 值可以提高查询子句的相对权重。因此我们可以像下面一样重写之前的查询：



```
GET /_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "content": {
              "query": "full text search",
              "operator": "and"
            }
          }
        }
      ],
      "should": [
        {
          "match": {
            "content": {
              "query": "Elasticsearch",
              "boost": 3 (1)
            }
          }
        },
        {
          "match": {
            "content": {
              "query": "Lucene",
              "boost": 2 (2)
            }
          }
        }
      ]
    }
  }
}
```

1. 这些查询子句的 `boost` 值为默认值 `1`。
2. 这个子句是最重要的，因为他有最高的 `boost` 值。
3. 这个子句比第一个查询子句的要重要，但是没有“`Elasticsearch`”子句重要。

注意：

1. `boost` 参数用于提高子句的相对权重（`boost` 值大于 `1`）或者降低子句的相对权重（`boost` 值在 `0 - 1` 之间），但是提高和降低并非是线性的。换句话说，`boost` 值为 `2` 并不能够使结果变成两部的得分。
2. 另外，`boost` 值被使用了以后新的得分是标准的。每个查询类型都会有一个独有的标准算法，算法的详细内容并不在本书的范畴。简单的概括一下，一个更大的 `boost` 值可以得到一个更高的得分。
3. 如果你自己实现了没有基于TF/IDF的得分模型，但是你想得到更多的对于提高得分过程的控制，你可以使用 `function_score` 查询来调整一个文档的 `boost` 值而不用通过标准的步骤。



我们会在下一章介绍更多的组合查询，[【multi-field-search】](#)。但是首先让我们一起来看一下查询的另外一个重要特征：文本分析。<!-- === 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 normalised 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 normalised 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. -->



分析控制

查询只能查找在倒排序索引中出现的词，所以确保在文档索引的时候以及字符串查询的时候使用同一个分析器是很重要的，为了查询的词能够在倒排序索引中匹配到。

尽管我们说文档中每个字段的分析器是已经定好的。但是字段可以有不同的分析器，通过给那个字段配置一个指定的分析器或者直接使用类型，索引，或节点上的默认分析器。在索引的时候，一个字段的值会被配置的或者默认的分析器分析。

举个例子，让我们添加一个字段到 `my_index` :

```
PUT /my_index/_mapping/my_type
{
  "my_type": {
    "properties": {
      "english_title": {
        "type": "string",
        "analyzer": "english"
      }
    }
  }
}
```

现在我们可以通过 `analyze API` 分析 `Foxes` 词来比较 `english_title` 字段中的值以及 `title` 字段中的值在创建索引的时候是怎么被分析的：

```
GET /my_index/_analyze?field=my_type.title <1>
Foxes

GET /my_index/_analyze?field=my_type.english_title <2>
Foxes
```

<1> 使用 `standard` 分析器的 `title` 字段将会返回词 `foxes`。

<2> 使用 `english` 分析器的 `english_title` 字段将会返回词 `fox`。

这意味着，如果我们为精确的词 `fox` 执行一个低级别的 `term` 查询，`english_title` 字段会匹配而 `title` 字段不会。

像 `match` 查询一样的高级别的查询可以知道字段的映射并且能够在被查询的字段上使用正确的分析器。我们可以在 `validate-query API` 的执行中看到这个：



```
GET /my_index/my_type/_validate/query?explain
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "Foxes" } },
        { "match": { "english_title": "Foxes" } }
      ]
    }
  }
}
```

它会返回 explanation :

```
(title:foxes english_title:fox)
```

match 查询为每个字段使用合适的分析器用来确保在那个字段上可以用正确的格式查询这个词。

默认分析器

虽然我们可以在字段级别指定一个分析器，但是如果我们在字段级别没有指定分析器的话，那要怎么决定某个字段使用什么分析器呢？

分析器可以在好几个地方设置。Elasticsearch会查找每个级别直到找到它可以使用的分析器。在创建索引的时候，Elasticsearch查找分析器的顺序如下：

- 在映射文件中指定字段的 analyzer，或者
- 在文档的 _analyzer 字段上指定分析器，或者
- 在映射文件中指定类型的默认分析器 analyzer
- 在索引映射文件中设置默认分析器 default
- 在节点级别设置默认分析器 default
- standard 分析器

查找示例的时候，Elasticsearch查找分析器的顺序稍微有点不一样：

- 在查询参数中指定 analyzer，或者
- 在映射文件中指定字段的 analyzer，或者
- 在映射文件中指定类型的默认分析器 analyzer
- 在索引映射文件中设置默认分析器 default
- 在节点级别设置默认分析器 default
- standard 分析器



提示：

上面列表中用斜体字的两行突出了创建索引以及查询索引的时候 Elasticsearch 查找分析器的区别。`_analyzer` 字段允许你为每个文档指定默认的分析器(比如, `english`, `french`, `spanish`)，虽然在查询的时候指定 `analyzer` 参数，但是在一个索引中处理多种语言这并不是一个好方法，因为在多语言环境下很多问题会暴露出来。

有时候，在创建索引与查询索引的时候使用不同的分析器也是有意义的。举个例子：在创建索引的时候想要索引同义词(比如, 出现`quick`的时候，我们也索引 `fast`, `rapid`, 和 `speedy`)。但是在查询索引的时候，我们不需要查询所有的同义词，我们只要查询用户输入的一个单词就可以了，它可以是 `quick` , `fast` , `rapid` , 或者 `speedy` 。

为了满足这种差异，Elasticsearch也支持 `index_analyzer` 和 `search_analyzer` 参数，并且分析器被命名为 `default_index` 和 `default_search` 。

把这些额外的参数考虑进去，Elasticsearch查找所有的分析器的顺序实际上像下面的样子：

- 在映射文件中指定字段的 `index_analyzer` , 或者
- 在映射文件中指定字段的 `analyzer` , 或者
- 在文档的 `_analyzer` 字段上指定分析器，或者
- 在映射文件中指定类型的创建索引的默认分析器 `index_analyzer`
- 在映射文件中指定类型的默认分析器 `analyzer`
- 在索引映射文件中设置创建索引的默认分析器 `default_index`
- 在索引映射文件中设置默认分析器 `default`
- 在节点级别设置创建索引的默认分析器 `default_index`
- 在节点级别设置默认分析器 `default`
- `standard` 分析器

以及查询索引的时候:

- 在查询参数中指定 `analyzer` , 或者
- 在映射文件中指定字段的 `search_analyzer` , 或者
- 在映射文件中指定字段的 `analyzer` , 或者
- 在映射文件中指定类型的查询索引的默认分析器 `analyzer`
- 在索引映射文件中设置查询索引的默认分析器 `default_search`
- 在索引映射文件中设置默认分析器 `default_search`
- 在节点级别设置查询索引的默认分析器 `default_search`
- 在节点级别设置默认分析器 `default`
- `standard` 分析器

实际配置分析器

可以指定分析器的地方实在是太多了，但实际上，指定分析器很简单。



用索引配置，而不是用配置文件

第一点要记住的是，尽管你开始使用 Elasticsearch 仅仅只是为了一个简单的目的或者是一个应用比如日志，但很可能你会发现更多的案例（use cases 翻译成案例不知道合不合适，如果有更好的用词，请联系我，Tks），结局是在同一个集群中运行着好几个不同的应用。每一个索引都需要是独立的，并且可以被独立的配置。你不要想着给一个案例设置默认值，但是不得不重写他们来适配后面的案例。

这个规则把节点级别的配置分析器方法排除在外了。另外，**节点级别的分析器配置方法需要改变每个节点的配置文件并且重启每个节点，这将成为维护的噩梦。保持 Elasticsearch 持续运行并通过 API 来管理索引的设置是一个更好的方法。**

保持简便性

大多数时间，你可以预先知道文档会包含哪些字段。**最简单的方法是在你创建索引或者添加类型映射的时候为每一个全文检索字段设置分析器。**虽然这个方法有点啰嗦，但是它可以很容易的看到哪个字段应用了哪个分析器。

通常情况下，大部分的字符串字段是确切值 not_analyzed 字段（索引但不分析字段）比如标签，枚举，加上少数全文检索字段会使用默认的分析器，像 standard 或者 english 或者其他语言。然后你可能只有一到两个字段需要定制分析：或许 title 字段需要按照你查找的方式去索引来支持你的查找。（指的是你查找的字符串用什么分析器，创建索引就用什么分析器）

你可以在索引设置 default 分析器的地方为几乎所有全文检索字段设置成你想要的分析器，并且只要在一到两个字段指定专门的分析器。如果，在你的模型中，你每个类型都需要不同的分析器，那么在类型级别使用 analyzer 配置来代替。

提示：

一个普通的像日志一样的基于时间轴的工作流数据每天都得创建新的索引，忙着不断的创建索引。虽然这种工作流阻止你预先创建索引，但是你可以使用索引模板来指定新的索引的配置和映射。



关联失效

在我们去讨论多字段检索中的更复杂的查询前，让我们顺便先解释一下为什么我们只用一个主分片来创建索引。

有时有的新手会问一个问题说通过相关性排序没有效果，并且提供了一小段复制的结果：该用户创建了一些文档，执行了一个简单的查询，结果发现相关性较低的结果排在了相关性较高的结果的前面。

为了理解为什么会出现这样的结果，我们假设用两个分片创建一个索引，以及索引10个文档，6个文档包含词 `foo`，这样可能会出现分片1中有3个文档包含 `foo`，分片2中也有三个文档包含 `foo`。换句话说，我们的文档做了比较好的分布式。

在相关性介绍这一节，我们描述了Elasticsearch默认的相似算法，叫做词频率/反转文档频率（TF/IDF）。词频率是一个词在我们当前查询的文档的字段中出现的次数。出现的次数越多，相关性就越大。反转文档频率指的是该索引中所有文档数与出现这个词的文件数的百分比，词出现的频率越大，IDF越小。

然而，由于性能问题，Elasticsearch不通过索引中所有的文档计算IDF。每个分片会为分片中所有的文档计算一个本地的IDF取而代之。

因为我们的文档做了很好的分布式，每个分片的IDF是相同的。现在假设5个包含 `foo` 的文档在分片1中，以及其他5个文档在分片2中。在这个场景下，词 `foo` 在第一个分片中是非常普通的（因此重要性较小），但是在另一个分片中是很稀少的（因此重要性较高）。这些区别在IDF中就会产生不正确的结果。

事实证明，这并不是一个问题。你索引越多的文档，本地IDF和全局IDF的区别就会越少。在实际工作的数据量下，本地IDF立刻能够很好的工作（With real-world volumes of data, the local IDFs soon even out，不知道这么翻译合不合适）。所以问题不是因为关联失效，而是因为数据太少。

为了测试的目的，对于这个问题，有两种方法可以奏效。第一种方法是创建一个只有一个主分片的索引，像我们介绍 `match` 查询那节一样做。如果你只有一个分片，那么本地IDF就是全局IDF。

第二种方法是在你们请求中添加 `?search_type=dfs_query_then_fetch`。`dfs` 就是*Distributed Frequency Search*，并且它会告诉Elasticsearch检查每一个分片的本地IDF为了计算整个索引的全局IDF。

提示：不要把 `dfs_query_then_fetch` 用于生产环境。它实在是没有必要。只要有足够的数据就能够确保词频率很好的分布。没有理由在每个你要执行的查询中添加额外的DFS步骤。



多字段搜索

只有一个简单的 `match` 子句的查询是很少的。我们经常需要在一个或者多个字段中查询相同的或者不同的查询字符串，意味着我们需要能够组合多个查询子句以及使他们的相关性得分有意义。

或许我们在寻找列夫·托尔斯泰写的一本叫《战争与和平》的书。或许我们在 Elasticsearch 的文档中查找 `minimum should match`，它可能在标题中，或者在一页的正文中。或许我们查找名为 John，姓为 Smith 的人。

在这一章节，我们会介绍用于构建多个查询子句搜索的可能的工具，以及怎么样选择解决方案来应用到你特殊的场景。



多重查询字符串

在明确的字段中的词查询是最容易处理的多字段查询。如果我们知道War and Peace是标题，Leo Tolstoy是作者，可以很容易的用match查询表达每个条件，并且用布尔查询组合起来：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "War and Peace" }},
        { "match": { "author": "Leo Tolstoy"   }}
      ]
    }
  }
}
```

布尔查询采用“匹配越多越好(More-matches-is-better)”的方法，所以每个match子句的得分会
被加起来变成最后的每个文档的得分。匹配两个子句的文档的得分会比只匹配了一个文档的
得分高。

当然，没有限制你只能使用match子句：布尔查询可以包装任何其他的查询类型，包含其他的
布尔查询，我们可以添加一个子句来指定我们更喜欢看被哪个特殊的翻译者翻译的那版书：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "match": { "title": "War and Peace" }},
        { "match": { "author": "Leo Tolstoy"   }},
        { "bool": {
          "should": [
            { "match": { "translator": "Constance Garnett" }},
            { "match": { "translator": "Louise Maude"   }}
          ]
        }}
      ]
    }
  }
}
```

为什么我们把翻译者的子句放在一个独立的布尔查询中？所有的匹配查询都是should子句，
所以为什么不把翻译者的子句放在和title以及作者的同一级？



答案就在如何计算得分中。布尔查询执行每个匹配查询，把他们的得分加在一起，然后乘以匹配子句的数量，并且除以子句的总数。每个同级的子句权重是相同的。在前面的查询中，包含翻译者的布尔查询占用总得分的三分之一。如果我们把翻译者的子句放在和标题与作者同级的目录中，我们会把标题与作者的作用减少的四分之一。

设置子句优先级

在先前的查询中我们可能不需要使每个子句都占用三分之一的权重。我们可能对标题以及作者比翻译者更感兴趣。我们需要调整查询来使得标题和作者的子句相关性更重要。

最简单的方法是使用boost参数。为了提高标题和作者字段的权重，我们给boost参数提供一个比1高的值：

```
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" } }
          ]
        } }
      ]
    }
  }
}
```

<1> 标题和作者的boost值为2。

<2> 嵌套的布尔查询的boost值为默认的1。

通过试错(Trial and Error)的方式可以确定"最佳"的boost值：设置一个boost值，执行测试查询，重复这个过程。一个合理boost值的范围在1和10之间，也可能是15。比它更高的值的影响不会起到很大的作用，因为分值会被规范化(Normalized)。



单一查询字符串(Single Query String)

bool查询是多字段查询的中流砥柱。在很多场合下它都能很好地工作，特别是当你能够将不同的查询字符串映射到不同的字段时。

问题在于，现在的用户期望能够在同一个地方输入所有的搜索词条，然后应用能够知道如何为他们得到正确的结果。所以当我们把含有多个字段的搜索表单称为高级搜索(Advanced Search)时，是有一些讽刺意味的。高级搜索虽然对用户而言会显得更"高级"，但是实际上它的实现方式更简单。

对于多词，多字段查询并没有一种万能(one-size-fits-all)的方法。要得到最佳的结果，你需要了解你的数据以及如何使用恰当的工具。

了解你的数据

当用户的唯一输入就是一个查询字符串时，你会经常碰到以下三种情况：

1. 最佳字段(Best fields)::

当搜索代表某些概念的单词时，例如"brown fox"，几个单词合在一起表达出来的意思比单独的单词更多。类似**title**和**body**的字段，尽管它们是相关联的，但是也是互相竞争着的。**文档在相同的字段中应该有尽可能多的单词(译注：搜索的目标单词)，文档的分数应该来自拥有最佳匹配的字段。**

2. 多数字段(Most fields)::

一个用来调优相关度的常用技术是**将相同的数据索引到多个字段中**，每个字段拥有自己的分析链(Analysis Chain)。

主要字段会含有单词的词干部分，同义词和消除了变音符号的单词。它用来尽可能多地匹配文档。

相同的文本可以被索引到其它的字段中来提供更加精确的匹配。**一个字段或许会包含未被提取成词干的单词，另一个字段是包含了变音符号的单词，第三个字段则使用shingle来提供关于单词邻近度(Word Proximity)的信息。**

以上这些额外的字段扮演者**signal**的角色，用来增加每个匹配的文档的相关度分值。越多的字段被匹配则意味着文档的相关度越高。

3. 跨字段(Cross fields)::

对于一些实体，标识信息会在多个字段中出现，每个字段中只含有一部分信息：

- Person: `first_name` 和 `last_name`



- Book: title , author , 和 description
- Address: street , city , country , 和 postcode

此时，我们希望在任意字段中找到尽可能多的单词。我们需要在多个字段中进行查询，就好像这些字段是一个字段那样。

以上这些都是多词，多字段查询，但是每种都需要使用不同的策略。我们会在本章剩下的部分解释每种策略。



最佳字段

假设我们有一个让用户搜索博客文章的网站(允许多字段搜索，最佳字段查询)，就像这两份文档一样：

```
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

用户输入了"Brown fox"，然后按下了搜索键。我们无法预先知道用户搜索的词条会出现在博文的**title**或者**body**字段中，但是用户是在搜索和他输入的单词相关的内容。右眼观察，以上的两份文档中，文档2似乎匹配的更好一些，因为它包含了用户寻找的两个单词。

让我们运行下面的**bool**查询：

```
{
    "query": {
        "bool": {
            "should": [
                { "match": { "title": "Brown fox" }},
                { "match": { "body": "Brown fox" }}
            ]
        }
    }
}
```

// SENSE: 110_Multi_Field_Search/15_Best_fields.json

然后我们发现文档1的分值更高：



```
{  
  "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."  
      }  
    }  
  ]  
}
```

要理解原因，想想bool查询是如何计算得到其分值的：

- 1.运行should子句中的两个查询
- 2.相加查询返回的分值
- 3.将相加得到的分值乘以匹配的查询子句的数量
- 4.除以总的查询子句的数量

文档1在两个字段中都包含了brown，因此两个match查询都匹配成功并拥有了一个分值。文档2在body字段中包含了brown以及fox，但是在title字段中没有出现任何搜索的单词。因此对body字段查询得到的高分加上对title字段查询得到的零分，然后在乘以匹配的查询子句数量1，最后除以总的查询子句数量2，导致整体分值比文档1的低。

在这个例子中，title和body字段是互相竞争的。我们想要找到一个最佳匹配(Best-matching)的字段。

如果我们不是合并来自每个字段的分值，而是使用最佳匹配字段的分值作为整个查询的整体分值呢？这就会让包含有我们寻找的两个单词的字段有更高的权重，而不是在不同的字段中重复出现的相同单词。

dis_max查询

相比使用bool查询，我们可以使用dis_max查询(Disjunction Max Query)。Disjunction的意思"OR"(而Conjunction的意思是"AND")，因此Disjunction Max Query的意思就是返回匹配了任何查询的文档，并且分值是产生了最佳匹配的查询所对应的分值：



```
{  
    "query": {  
        "dis_max": {  
            "queries": [  
                { "match": { "title": "Brown fox" }},  
                { "match": { "body": "Brown fox" }}  
            ]  
        }  
    }  
}
```

// SENSE: 110_Multi_Field_Search/15_Best_fields.json

它会产生我们期望的结果：

```
{  
    "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."  
            }  
        }  
    ]  
}
```



最佳字段查询的调优

如果用户(((“multifield search”, “best fields queries”, “tuning”))))(((“best fields queries”, “tuning”)))搜索的是“quick pets”，那么会发生什么呢？两份文档都包含了单词quick，但是只有文档2包含了单词pets。两份文档都没能在同一个字段中同时包含搜索的两个单词。

一个像下面那样的简单dis_max查询会选择出拥有最佳匹配字段的查询子句，而忽略其他的查询子句：

```
{  
    "query": {  
        "dis_max": {  
            "queries": [  
                { "match": { "title": "Quick pets" }},  
                { "match": { "body": "Quick pets" }}  
            ]  
        }  
    }  
}
```

// SENSE: 110_Multi_Field_Search/15_Best_fields.json

```
{  
    "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> 可以发现，两份文档的分值是一模一样的。

我们期望的是同时匹配了title字段和body字段的文档能够拥有更高的排名，但是结果并非如此。需要记住：dis_max查询只是简单的使用最佳匹配查询子句得到的_score。



tie_breaker

但是，将其它匹配的查询子句考虑进来也是可能的。通过指定tie_breaker参数：

```
{  
    "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

它会返回以下结果：

```
{  
    "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> 现在文档2的分值比文档1稍高一些。

tie_breaker参数会让dis_max查询的行为更像是dis_max和bool的一种折中。它会通过下面的方式改变分值计算过程：

- 1.取得最佳匹配查询子句的_score。
- 2.将其它每个匹配的子句的分值乘以tie_breaker。



- 3.将以上得到的分值进行累加并规范化。

通过tie_breaker参数，所有匹配的子句都会起作用，只不过最佳匹配子句的作用更大。

提示：tie_breaker的取值范围是0到1之间的浮点数，取0时即为仅使用最佳匹配子句(译注：和不使用tie_breaker参数的dis_max查询效果相同)，取1则会将所有匹配的子句一视同仁。它的的确切值需要根据你的数据和查询进行调整，但是一个合理的值会靠近0，(比如，0.1 -0.4)，来确保不会压倒dis_max查询具有的最佳匹配性质。



multi_match查询

multi_match查询提供了一个简便的方法用来对多个字段执行相同的查询。

提示：存在几种类型的multi_match查询，其中的3种正好和在“单一查询字符串”小节中“了解你的数据”单元中提到的几种类型相同：best_fields，most_fields以及cross_fields。

默认情况下，该查询以best_fields类型执行，它会为每个字段生成一个match查询，然后将这些查询包含在一个dis_max查询中。下面的dis_max查询：

```
{  
  "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  
  }  
}
```

可以通过multi_match简单地重写如下：

```
{  
  "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> 注意到以上的type属性为best_fields。

<2> minimum_should_match和operator参数会被传入到生成的match查询中。

在字段名中使用通配符

字段名可以通过通配符指定：任何匹配了通配符的字段都会被包含在搜索中。你可以通过下面的查询来匹配book_title，chapter_title以及section_title字段：

```
{
  "multi_match": {
    "query": "Quick brown fox",
    "fields": "*_title"
  }
}
```

加权个别字段

个别字段可以通过caret语法(^)进行加权：仅需要在字段名后添加^boost，其中的boost是一个浮点数：

```
{
  "multi_match": {
    "query": "Quick brown fox",
    "fields": [ "*_title", "chapter_title^2" ] <1>
  }
}
```

<1> chapter_title字段的boost值为2，而book_title和section_title字段的boost值为默认的1。



多数字段(Most Fields)

全文搜索是一场召回率(Recall) - 返回所有相关的文档，以及准确率(Precision) - 不返回无关文档，之间的战斗。目标是在结果的第一页给用户呈现最相关的文档。

为了提高召回率，我们会广撒网 - 不仅包括精确匹配了用户搜索词条的文档，还包括了那些我们认为和查询相关的文档。如果一个用户搜索了"quick brown fox"，一份含有fast foxes的文档也可以作为一个合理的返回结果。

如果我们拥有的相关文档仅仅是含有fast foxes的文档，那么它会出现在结果列表的顶部。但是如果我们将100份含有quick brown fox的文档，那么含有fast foxes的文档的相关性就会变低，我们希望它出现在结果列表的后面。在包含了许多可能的匹配后，我们需要确保相关度高的文档出现在顶部。

一个用来调优全文搜索相关性的常用技术是将同样的文本以多种方式索引，每一种索引方式都提供了不同相关度的信号(Signal)。主要字段(Main field)中含有的词条的形式是最宽泛的(Broadcast-matching)，用来尽可能多的匹配文档。比如，我们可以这样做：

- 使用一个词干提取器来将jumps，jumping和jumped索引成它们的词根：jump。然后当用户搜索的是jumped时，我们仍然能够匹配含有jumping的文档。
- 包含同义词，比如jump，leap和hop。
- 移除变音符号或者声调符号：比如，ésta，está和esta都会以esta被索引。但是，如果我们有两份文档，其中之一含有jumped，而另一份含有jumping，那么用户会希望第一份文档的排序会靠前，因为它含有用户输入的精确值。

我们可以通过将相同的文本索引到其它字段来提供更加精确的匹配。一个字段可以包含未被提取词干的版本，另一个则是含有变音符号的原始单词，然后第三个使用了shingles，用来提供和单词邻近度相关的信息。这些其它字段扮演的角色就是信号(Signals)，它们用来增加每个匹配文档的相关度分值。能够匹配的字段越多，相关度就越高。

如果一份文档能够匹配具有最宽泛形式的主要字段(Main field)，那么它就会被包含到结果列表中。如果它同时也匹配了信号字段，它会得到一些额外的分值用来将它移动到结果列表的前面。

我们会在本书的后面讨论同义词，单词邻近度，部分匹配以及其他可能的信号，但是我们会使用提取了词干和未提取词干的字段的简单例子来解释这个技术。

多字段映射(Multifield Mapping)

第一件事就是将我们的字段索引两次：一次是提取了词干的形式，一次是未提取词干的形式。为了实现它，我们会使用多字段(Multifields)，在字符串排序和多字段中我们介绍过：



```
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 <<关联失效(相关性被破坏)>>.

<2> title字段使用了english解析器进行词干提取。

<3> title.std字段则使用的是standard解析器，因此它没有进行词干提取。

下一步，我们会索引一些文档：

```
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

以下是一个简单的针对title字段的match查询，它查询jumping rabbits：



```
GET /my_index/_search
{
  "query": {
    "match": {
      "title": "jumping rabbits"
    }
  }
}
```

// SENSE: 110_Multi_Field_Search/30_Most_fields.json

它会变成一个针对两个提干后的词条jump和rabbit的查询，这要得益于english解析器。两份文档的title字段都包含了以上两个词条，因此两份文档的分值是相同的：

```
{
  "hits": [
    {
      "_id": "1",
      "_score": 0.42039964,
      "_source": {
        "title": "My rabbit jumps"
      }
    },
    {
      "_id": "2",
      "_score": 0.42039964,
      "_source": {
        "title": "Jumping jack rabbits"
      }
    }
  ]
}
```

如果我们只查询title.std字段，那么只有文档2会匹配。但是，当我们查询两个字段并将它们的分值通过bool查询进行合并的话，两份文档都能够匹配(title字段也匹配了)，而文档2的分值会更高一些(匹配了title.std字段)：

```
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> 在上述查询中，由于我们想合并所有匹配字段的分值，因此使用的类型为most_fields。这会让multi_match查询将针对两个字段的查询子句包含在一个bool查询中，而不是包含在一个dis_max查询中。

```
{  
  "hits": [  
    {  
      "_id": "2",  
      "_score": 0.8226396, <1>  
      "_source": {  
        "title": "Jumping jack rabbits"  
      }  
    },  
    {  
      "_id": "1",  
      "_score": 0.10741998, <1>  
      "_source": {  
        "title": "My rabbit jumps"  
      }  
    }  
  ]  
}
```

<1> 文档2的分值比文档1的高许多。

我们使用了拥有宽泛形式的title字段来匹配尽可能多的文档 - 来增加召回率(Recall)，同时也使用了title.std字段作为信号来让最相关的文档能够拥有更靠前的排序(译注：增加了准确率(Precision))。

每个字段对最终分值的贡献可以通过指定boost值进行控制。比如，我们可以提升title字段来让该字段更加重要，这也减小了其它信号字段的影响：

```
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> boost=10让title字段的相关性比title.std更重要。



跨字段实体搜索(Cross-fields Entity Search)

现在让我们看看一个常见的模式：跨字段实体搜索。类似person，product或者address这样的实体，它们的信息会分散到多个字段中。我们或许有一个person实体被索引如下：

```
{  
    "firstname": "Peter",  
    "lastname": "Smith"  
}
```

而address实体则是像下面这样：

```
{  
    "street": "5 Poland Street",  
    "city": "London",  
    "country": "United Kingdom",  
    "postcode": "W1V 3DG"  
}
```

这个例子也许很像在多查询字符串中描述的，但是有一个显著的区别。在多查询字符串中，
我们对每个字段都使用了不同的查询字符串。在这个例子中，我们希望使用一个查询字符串
来搜索多个字段。

用户也许会搜索名为"Peter Smith"的人，或者名为"Poland Street W1V"的地址。每个查询的
单词都出现在不同的字段中，因此使用dis_max/best_fields查询来搜索单个最佳匹配字段显然
是不对的。

一个简单的方法

实际上，我们想要依次查询每个字段然后将每个匹配字段的分值进行累加，这听起来很像bool
查询能够胜任的工作：

```
{  
    "query": {  
        "bool": {  
            "should": [  
                { "match": { "street": "Poland Street W1V" }},  
                { "match": { "city": "Poland Street W1V" }},  
                { "match": { "country": "Poland Street W1V" }},  
                { "match": { "postcode": "Poland Street W1V" }}  
            ]  
        }  
    }  
}
```



对每个字段重复查询字符串很快就会显得冗长。我们可以使用multi_match查询进行替代，然后将type设置为most_fields来让它将所有匹配字段的分值合并：

```
{  
  "query": {  
    "multi_match": {  
      "query": "Poland Street W1V",  
      "type": "most_fields",  
      "fields": [ "street", "city", "country", "postcode" ]  
    }  
  }  
}
```

使用**most_fields**存在的问题

使用most_fields方法执行实体查询有一些不那么明显的问题：

- 它被设计用来找到匹配任意单词的多数字段，而不是找到跨越所有字段的最匹配的单词。
- 它不能使用operator或者minimum_should_match参数来减少低相关度结果带来的长尾效应。
- 每个字段的词条频度是不同的，会互相干扰最终得到较差的排序结果。



以字段为中心的查询(Field-centric Queries)

上述提到的三个问题都来源于`most_fields`是以字段为中心(Field-centric)，而不是以词条为中心(Term-centric)：它会查询最多匹配的字段(Most matching fields)，而我们真正感兴趣的最匹配的词条(Most matching terms)。

提示：`best_fields`同样是以字段为中心的，因此它也存在相似的问题。首先我们来看看为什么存在这些问题，以及如何解决它们。

问题1：在多个字段中匹配相同的单词

考虑一下`most_fields`查询是如何执行的：ES会为每个字段生成一个match查询，然后将它们包含在一个bool查询中。

我们可以将查询传入到validate-query API中进行查看：

```
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

它会产生下面的解释(explaination)：

```
(street:poland street:street street:w1v)
(city:poland city:street city:w1v)
(country:poland country:street country:w1v)
(postcode:poland postcode:street postcode:w1v)
```

你可以发现能够在两个字段中匹配poland的文档会比在一个字段中匹配了poland和street的文档的分值要高。

问题2：减少长尾

在精度控制(Controlling Precision)一节中，我们讨论了如何使用and操作符和`minimum_should_match`参数来减少相关度低的文档数量：



```
{  
    "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> 所有的term必须存在。

但是，使用best_fields或者most_fields，这些参数会被传递到生成的match查询中。该查询的解释如下(译注：通过validate-query API)：

```
(+street:poland +street:street +street:w1v)  
(+city:poland +city:street +city:w1v)  
(+country:poland +country:street +country:w1v)  
(+postcode:poland +postcode:street +postcode:w1v)
```

换言之，使用and操作符时，所有的单词都需要出现在相同的字段中，这显然是错的！这样做可能不会有任何匹配的文档。

问题3：词条频度

在[什么是相关度\(What is Relevance\(relevance-intro\)\)](#)一节中，我们解释了默认用来计算每个词条的相关度分值的相似度算法TF/IDF：

- 词条频度(Term Frequency)::

在一份文档中，一个词条在一个字段中出现的越频繁，文档的相关度就越高。

- 倒排文档频度(Inverse Document Frequency)::

一个词条在索引的所有文档的字段中出现的越频繁，词条的相关度就越低。当通过多字段进行搜索时，TF/IDF会产生一些令人惊讶的结果。

考虑使用first_name和last_name字段搜索"Peter Smith"的例子。Peter是一个常见的名字，Smith是一个常见的姓氏 - 它们的IDF都较低。但是如果在索引中有另外一个名为Smith Williams的人呢？Smith作为名字是非常罕见的，因此它的IDF值会很高！

像下面这样的一个简单查询会将Smith Williams放在Peter Smith前面(译注：含有Smith Williams的文档分值比含有Peter Smith的文档分值高)，尽管Peter Smith明显是更好的匹配：



```
{  
    "query": {  
        "multi_match": {  
            "query": "Peter Smith",  
            "type": "most_fields",  
            "fields": ["*_name"]  
        }  
    }  
}
```

// SENSE: 110_Multi_Field_Search/40_Bad_frequencies.json

smith在first_name字段中的高IDF值会压倒peter在first_name字段和smith在last_name字段中的两个低IDF值。

解决方案

这个问题仅在我们处理多字段时存在。如果我们将所有这些字段合并到一个字段中，该问题就不复存在了。我们可以向person文档中添加一个full_name字段来实现：

```
{  
    "first_name": "Peter",  
    "last_name": "Smith",  
    "full_name": "Peter Smith"  
}
```

当我们只查询full_name字段时：

- 拥有更多匹配单词的文档会胜过那些重复出现一个单词的文档。
- minimum_should_match和operator参数能够正常工作。
- first_name和last_name的倒排文档频度会被合并，因此smith无论是first_name还是last_name都不再重要。尽管这种方法能工作，可是我们并不想存储冗余数据。因此，ES为我们提供了两个解决方案 - 一个在索引期间，一个在搜索期间。下一节对它们进行讨论。



自定义_all字段

在元数据：`_all`字段中，我们解释了特殊的`_all`字段会将其它所有字段中的值作为一个大字符串进行索引。尽管将所有字段的值作为一个字段进行索引并不是非常灵活。如果有一个自定义的`_all`字段用来索引人名，另外一个自定义的`_all`字段用来索引地址就更好了。

ES通过字段映射中的`copy_to`参数向我们提供了这一功能：

```
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>`first_name`和`last_name`字段中的值会被拷贝到`full_name`字段中。

有了这个映射，我们可以通过`first_name`字段查询名字，`last_name`字段查询姓氏，或者`full_name`字段查询姓氏和名字。

提示：`first_name`和`last_name`字段的映射和`full_name`字段的索引方式的无关。

`full_name`字段会从其它两个字段中拷贝字符串的值，然后仅根据`full_name`字段自身的映射进行索引。



跨域查询(Cross-fields Queries)

如果你在索引文档前就能够自定义_all字段的话，那么使用_all字段就是一个不错的方法。但是，ES同时也提供了一个搜索期间的解决方案：使用类型为cross_fields的multi_match查询。cross_fields类型采用了一种以词条为中心(Term-centric)的方法，这种方法和best_fields及most_fields采用的以字段为中心(Field-centric)的方法有很大的区别。它将所有的字段视为一个大的字段，然后在任一字段中搜索每个词条。

为了阐述以字段为中心和以词条为中心的查询的区别，看看以字段为中心的most_fields查询的解释(译注：通过validate-query API得到)：

```
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>operator设为了and，表示所有的词条都需要出现。

对于一份匹配的文档，peter和smith两个词条都需要出现在相同的字段中，要么是first_name字段，要么是last_name字段：

```
(+first_name:peter +first_name:smith)
(+last_name:peter +last_name:smith)
```

而以词条为中心的方法则使用了下面这种逻辑：

```
+(first_name:peter last_name:peter)
+(first_name:smith last_name:smith)
```

换言之，词条peter必须出现在任一字段中，同时词条smith也必须出现在任一字段中。

cross_fields类型首先会解析查询字符串来得到一个词条列表，然后在任一字段中搜索每个词条。仅这个区别就能够解决在以字段为中心的查询中提到的3个问题中的2个，只剩下倒排文档频度的不同这一问题。



幸运的是，cross_fields类型也解决了这个问题，从下面的validate-query请求中可以看到：

```
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> cross_fields 使用以词条为中心(Term-centric)进行匹配。

它通过混合(Blending)字段的倒排文档频度来解决词条频度的问题：

```
+blended("peter", fields: [first_name, last_name])
+blended("smith", fields: [first_name, last_name])
```

换言之，**它会查找词条smith在first_name和last_name字段中的IDF值，然后使用两者中较小的作为两个字段最终的IDF值**。因为smith是一个常见的姓氏，意味着它也会被当做一个常见的名字。

提示：为了让cross_fields查询类型能以最佳的方式工作，所有的字段都需要使用相同的解析器。使用了相同的解析器的字段会被组合在一起形成混合字段(Blended Fields)。

如果你包含了使用不同解析链(Analysis Chain)的字段，它们会以和best_fields相同的方式被添加到查询中。比如，如果我们将title字段添加到之前的查询中(假设它使用了一个不同的解析器)，得到的解释如下所示：

```
(+title:peter +title:smith)
(
  +blended("peter", fields: [first_name, last_name])
  +blended("smith", fields: [first_name, last_name])
)
```

当使用了minimum_should_match以及operator参数时，这一点尤为重要。

逐字段加权(Per-field Boosting)



使用cross_fields查询相比使用自定义_all字段的一个优点是你能够在查询期间对个别字段进行加权。

对于first_name和last_name这类拥有近似值的字段，也许加权是不必要的，但是如果你通过title和description字段来搜索书籍，那么你或许会给予title字段更多的权重。这可以通过前面介绍的caret(^)语法来完成：

```
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 .

能够对个别字段进行加权带来的优势应该和对多个字段执行查询伴随的代价进行权衡，因为如果使用自定义的_all字段，那么只需要对一个字段进行查询。选择能够给你带来最大收益的方案。



精确值字段(Exact-value Fields)

在结束对于多字段查询的讨论之前的最后一个话题是作为not_analyzed类型的精确值字段。在multi_match查询中将not_analyzed字段混合到analyzed字段中是没有益处的。

原因可以通过validate-query进行简单地验证，假设我们将title字段设置为not_analyzed：

```
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

因为title字段时没有被解析的，它会以将整个查询字符串作为一个词条进行搜索！

```
title:peter smith
(
  blended("peter", fields: [first_name, last_name])
  blended("smith", fields: [first_name, last_name])
)
```

很显然该词条在title字段的倒排索引中并不存在，因此永远不可能被找到。在multi_match查询中避免使用not_analyzed字段。



模糊匹配

一般的全文检索方式使用 TF/IDF 处理文本或者文本数据中的某个字段内容。将字面切分成很多字、词(word)建立索引，match查询用query中的term来匹配索引中的字、词。match查询提供了文档数据中是否包含我们需要的query中的单、词，但仅仅这样是不够的，它无法提供文本中的字词之间的关系。

举个例子：

- 小苏吃了鳄鱼
- 鳄鱼吃了小苏
- 小苏去哪儿都带着的鳄鱼皮钱包

用 match 查询 小苏 鳄鱼 ，这三句话都会被命中，但是 tf/idf 并不会告诉我们这两个词出现在同一句话里面还是在同一个段落中（仅仅提供这两个词在这段文本中的出现频率）

理解文本中词语之间的关系是一个很复杂的问题，而且这个问题通过更换query的表达方式是无法解决的。但是我们可以知道两个词语在文本中的距离远近，甚至是否相邻，这个信息似乎上能一定程度的表达这两个词比较相关。

一般的文本可能比我们举的例子长很多，正如我们提到的： 小苏 跟 鳄鱼 这两个词可能分布在文本的不同段落中。我们还是期望能找到这两个词分布均匀的文档，但是我们把这两个词距离比较近的文档赋予更好的相关性权重。

这就是段落匹配 (*phrase matching*) 或者模糊匹配 (*proximity matching*) 所做的事情。

【提示】

这一章，我们会用之之前在< match-test-data, match query > 中使用的文档做例子。



[[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”))) (((“matchphrase 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(((“matchphrase 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/\[Lucene nightly benchmarks\]](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 same 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 multfield (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" } },  
<2> { "_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 `b1`

If you were to run this query through the `validate-query` API, it would produce this explanation:

```
"johnnie walker b1*"
```



Like the `match_phrase` query, it accepts a `slop` parameter (see <>) to make the word order and relative positions ((("slop parameter", "match_phrase_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 `b1` and keep collecting terms (in alphabetical order) until it either runs out of terms with prefix `b1`, 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" } } } }
```

```
}
```

// 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

Rindfleischetikettierungsaufgabenübertragungsgesetz:: 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 `Aussprachewörterbuch`" 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-organisation,` but it also matches `Militär-__ges__-chichte` and `Rindfleischetikettierungüberwachungsaufgabenübertragungs-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:

$$\dots \text{tf}(t \text{ in } d) = \sqrt{\text{frequency } <1>} \dots$$

<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:

$$\dots \text{idf}(t) = 1 + \log (\text{numDocs} / (\text{docFreq} + 1)) <1> \dots$$



<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 / $\sqrt{\text{numTerms}}$ <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> 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`—` `[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> · $\sum (\langle 4 \rangle tf(t \text{ in } d) \langle 5 \rangle \cdot idf(t)^2 \langle 6 \rangle \cdot t.getBoost() \langle 7 \rangle \cdot norm(t,d) \langle 8 \rangle) (t \text{ in } q) \langle 4 \rangle$

<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")))



[[查询时提升]] === 查询时提升

在 <> 中，我们解释了 (((("relevance", "controlling", "query time boosting"))))(((("boosting", "query-time")))) 你可以怎样在查询时使用 `boost` 来使得一个查询项比其它的更重要。例如：

[source,json]

```
GET /_search { "query": { "bool": { "should": [ { "match": { "title": { "query": "quick brown fox", "boost": 2 <1> } } }, { "match": { <2> "content": "quick brown fox" } } ] } }}
```

}

<1> 查询项 `title` 的重要性是查询项 `content` 的 2 倍，因为它被因数 2 提升了。

<2> 没有 `boost` 值的查询项会拥有一个默认因数为 1 的提升。

查询时提升 是用于调节相关性的主要工具。任何类型的查询都接受 `boost` 参数。(((("boost parameter", "setting value")))) 把 `boost` 设置为 2 并不会简单的加倍最后的 `_score`；实际使用的 `boost` 值取决于标准化和一些内置的优化。然而，它确实意味着 `boost` 值为 2 的项的重要性是 `boost` 值为 1 的项的 2 倍。

事实上，对于一个实际的查询项，没有简单的算法来决定“正确”的 `boost` 值，它是边做边看的事。要记得 `boost` 仅仅是影响相关性分数的因素之一；它必须与其它因素竞争。例如，在之前的例子里，`title` 字段相较于 `content` 字段，可能已经有了一个“自然的”提升，这归功于 (((("field-length norm")))) <> (`title` 通常比相关的 `content` 要短)，所以，不要仅仅因为你觉得它应该被提升就盲目的提升一个字段应用一个提升并且检查结果。改变提升并且重新检查。

==== 提升一个索引

当在多个索引间搜索时，(((("boosting", "query-time", "boosting an index"))))(((("indices", "boosting an index")))) 你可以通过 `indices_boost` 参数提升这些索引中的某一个索引。(((("indices_boost parameter")))) 下面的例子中使用了这种方法，使得最近的索引文档拥有更高的权重：

[source,json]

```
GET /docs2014*/_search <1> { "indices_boost": { <2> "docs_2014_10": 3, "docs_2014_09": 2 }, "query": { "match": { "text": "quick brown fox" } } }
```

}



<1> 该多索引搜索包含了所有以 `docs_2014_` 开头的索引.

<2> 索引 `docs_2014_10` 中的文档将被因数 `3` 提升, `docs_2014_09` 中的文档被因数 `2` 提升, 其它匹配的索引将被默认的因数 `1` 提升.

===== `t.getBoost()`

boost 值可以通过 `<> t.getBoost()` 获得. (((("practical scoring function", "t.getBoost() method")))((("boosting", "query-time", "t.getBoost()"))))((("t.getBoost() method")))) 提升不会被应用在出现查询 DSL 的地方. 而是任何 boost 值都会被合并、传递到单独的 terms 中.

`t.getBoost()` 方法会返回任意应用到 term 本身或更高阶查询链的 `boost` 值.

[TIP]

事实上, 阅读 `<>` 输出略为复杂. 你不会在 `explanation` 看到它提到过 `boost` 值或 `t.getBoost()`. 提升是被放入了应用于特殊term的`<>`中. 尽管我们说 `queryNorm` 对于每一个 term 都是相同的, 但是你会发现已提升的term的 `queryNorm` 要比未提升的term 的 `queryNorm` 要高.

=====



[[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/1DlqSAK>[`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` documentation](http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-function-score-query.html#_field_value_factor).

==== 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 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. 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 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 (((("functionscore 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` function 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": 51.5, "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 <4> 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 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.



[[language-intro]] == Getting Started with Languages

Elasticsearch ships with a collection of language analyzers that provide good, basic, out-of-the-box (((language analyzers)))(((languages", "getting started with")))support for many of the world's most common languages:

Arabic, Armenian, Basque, Brazilian, Bulgarian, Catalan, Chinese, Czech, Danish, Dutch, English, Finnish, French, Galician, German, Greek, Hindi, Hungarian, Indonesian, Irish, Italian, Japanese, Korean, Kurdish, Norwegian, Persian, Portuguese, Romanian, Russian, Spanish, Swedish, Turkish, and Thai.

These analyzers typically(((language analyzers", "roles performed by"))) perform four roles:

- Tokenize text into individual words: + `The quick brown foxes` -> [`The` , `quick` , `brown` , `foxes`]
- Lowercase tokens: + `The` -> `the`
- Remove common *stopwords*: + [`The` , `quick` , `brown` , `foxes`] -> [`quick` , `brown` , `foxes`]
- Stem tokens to their root form: + `foxes` -> `fox`

Each analyzer may also apply other transformations specific to its language in order to make words from that(((language analyzers", "other transformations specific to the language"))) language more searchable:

- The `english` analyzer (((english analyzer)))removes the possessive 's : + `John's` -> `john`
- The `french` analyzer (((french analyzer)))removes *elisions* like `l'` and `qu'` and *diacritics* like `^` or `~` : + `l'église` -> `eglise`
- The `german` analyzer normalizes(((german analyzer))) terms, replacing `ä` and `ae` with `a` , or `ß` with `ss` , among others: + `äußerst` -> `ausserst`



[[using-language-analyzers]] === Using Language Analyzers

The built-in language analyzers are available globally and don't need to be configured before being used.((("language analyzers", "using"))) They can be specified directly in the field mapping:

[source,js]

```
PUT /my_index { "mappings": { "blog": { "properties": { "title": { "type": "string", "analyzer": "english" } } } } }
```

```
}
```

<1> The `title` field will use the `english` analyzer instead of the default `standard` analyzer.

Of course, by passing ((("english analyzer", "information lost with"))text through the `english` analyzer, we lose information:

[source,js]

```
GET /my_index/_analyze?field=title <1>
```

I'm not happy about the foxes

<1> Emits token: `i'm`, `happi`, `about`, `fox`

We can't tell if the document mentions one `fox` or many `foxes`; the word `not` is a stopword and is removed, so we can't tell whether the document is happy about foxes or not. By using the `english` analyzer, we have increased recall as we can match more loosely, but we have reduced our ability to rank documents accurately.

To get the best of both worlds, we can use `<>` to index the `title` field twice: once(((("multifields", "using to index a field with two different analyzers")))) with the `english` analyzer and once with the `standard` analyzer:

[source,js]



```
PUT /my_index { "mappings": { "blog": { "properties": { "title": { <1> "type": "string", "fields": { "english": { <2> "type": "string", "analyzer": "english" } } } } }}
```

}

<1> The main `title` field uses the `standard` analyzer.

<2> The `title.english` subfield uses the `english` analyzer.

With this mapping in place, we can index some test documents to demonstrate how to use both fields at query time:

[source,js]

```
PUT /my_index/blog/1 { "title": "I'm happy for this fox" }
```

```
PUT /my_index/blog/2 { "title": "I'm not happy about my fox problem" }
```

```
GET /_search { "query": { "multi_match": { "type": "most_fields", <1> "query": "not happy foxes", "fields": [ "title", "title.english" ] } } }
```

}

<1> Use the `<>` query type to match the same text in as many fields as possible.

Even (((“most fields queries”)))though neither of our documents contain the word `foxes` , both documents are returned as results thanks to the word stemming on the `title.english` field. The second document is ranked as more relevant, because the word `not` matches on the `title` field.



[[configuring-language-analyzers]] === Configuring Language Analyzers

While the language analyzers can be used out of the box without any configuration, most of them ((("english analyzer", "configuring")))((("language analyzers", "configuring"))))do allow you to control aspects of their behavior, specifically:

[[stem-exclusion]] Stem-word exclusion:: + Imagine, for instance, that users searching for((("language analyzers", "configuring", "stem word exclusion"))))((("stemming words", "stem word exclusion, configuring")))) the `World Health Organization` are instead getting results for organ health." The reason for this confusion is that both `organ` and `organization` are stemmed to the same root word: `organ`. Often this isn't a problem, but in this particular collection of documents, this leads to confusing results. We would like to prevent the words `organization` and `organizations` from being stemmed.

Custom stopwords::

The default list of stopwords((("stopwords", "configuring for language analyzers")))) used in English are as follows: + a, an, and, are, as, at, be, but, by, for, if, in, into, is, it, no, not, of, on, or, such, that, the, their, then, there, these, they, this, to, was, will, with + The unusual thing about `no` and `not` is that they invert the meaning of the words that follow them. Perhaps we decide that these two words are important and that we shouldn't treat them as stopwords.

To customize the behavior of the `english` analyzer, we need to create a custom analyzer that uses the `english` analyzer as its base but adds some configuration:

[source,js]

```
PUT /my_index { "settings": { "analysis": { "analyzer": { "my_english": { "type": "english", "stem_exclusion": [ "organization", "organizations" ], <1> "stopwords": [ <2> "a", "an", "and", "are", "as", "at", "be", "but", "by", "for", "if", "in", "into", "is", "it", "of", "on", "or", "such", "that", "the", "their", "then", "there", "these", "they", "this", "to", "was", "will", "with" ] } } } }}
```

```
GET /my_index/_analyze?analyzer=my_english <3>
```

The World Health Organization does not sell organs.

<1> Prevents `organization` and `organizations` from being stemmed

<2> Specifies a custom list of stopwords



<3> Emits tokens `world` , `health` , `organization` , `does` , `not` , `sell` , `organ`

We discuss stemming and stopwords in much more detail in <> and <>, respectively.



[[language-pitfalls]] === Pitfalls of Mixing Languages

If you have to deal with only a single language,(("languages", "mixing, pitfalls of")) count yourself lucky. Finding the right strategy for handling documents written in several languages can be challenging.(("indexing", "mixed languages, pitfalls of"))

==== At Index Time

Multilingual documents come in three main varieties:

- One predominant language per *document*, which may contain snippets from other languages (See <>.)
- One predominant language per *field*, which may contain snippets from other languages (See <>.)
- A mixture of languages per field (See <>.)

The goal, although not always achievable, should be to keep languages separate. Mixing languages in the same inverted index can be problematic.

===== Incorrect stemming

The stemming rules for German are different from those for English, French, Swedish, and so on.(("stemming words", "incorrect stemming in multilingual documents")) Applying the same stemming rules to different languages will result in some words being stemmed correctly, some incorrectly, and some not being stemmed at all. It may even result in words from different languages with different meanings being stemmed to the same root word, conflating their meanings and producing confusing search results for the user.

Applying multiple stemmers in turn to the same text is likely to result in rubbish, as the next stemmer may try to stem an already stemmed word, compounding the problem.

[[different-scripts]] .Stemmer per Script

The one exception to the *only-one-stemmer* rule occurs when each language is written in a different script. For instance, in Israel it is quite possible that a single document may contain Hebrew, Arabic, Russian (Cyrillic), and English:

הזהר איזה - Предупреждение - خذير - Warning

Each language uses a different script, so the stemmer for one language will not interfere with another, allowing multiple stemmers to be applied to the same text.



===== Incorrect inverse document frequencies

In <>, we explained that the more frequently a term appears in a collection of documents, the less weight that term has.((("inverse document frequency", "incorrect, in multilingual documents"))) For accurate relevance calculations, you need accurate term-frequency statistics.

A short snippet of German appearing in predominantly English text would give more weight to the German words, given that they are relatively uncommon. But mix those with documents that are predominantly German, and the short German snippets now have much less weight.

===== At Query Time

It is not sufficient just to think about your documents, though.((("queries", "mixed languages and"))) You also need to think about how your users will query those documents. Often you will be able to identify the main language of the user either from the language of that user's chosen interface (for example, `mysite.de` versus `mysite.fr`) or from the [HTTP header from the user's browser](http://bit.ly/1BwEl61[`accept-language`]).

User searches also come in three main varieties:

- Users search for words in their main language.
- Users search for words in a different language, but expect results in their main language.
- Users search for words in a different language, and expect results in that language (for example, a bilingual person, or a foreign visitor in a web cafe).

Depending on the type of data that you are searching, it may be appropriate to return results in a single language (for example, a user searching for products on the Spanish version of the website) or to combine results in the identified main language of the user with results from other languages.

Usually, it makes sense to give preference to the user's language. An English-speaking user searching the Web for ``deja vu'' would probably prefer to see the English Wikipedia page rather than the French Wikipedia page.

[[identifying-language]] ===== Identifying Language

You may already know the language of your documents. Perhaps your documents are created within your organization and translated into a list of predefined languages. Human pre-identification is probably the most reliable method of classifying language correctly.



Perhaps, though, your documents come from an external source without any language classification, or possibly with incorrect classification. In these cases, you need to use a heuristic to identify the predominant language. Fortunately, libraries are available in several languages to help with this problem.

Of particular note is the [http://bit.ly/1AUr3i2\[chromium-compact-language-detector\]](http://bit.ly/1AUr3i2) library from <http://bit.ly/1AUr85k>[Mike McCandless], which uses the open source (<http://bit.ly/1u9KKgI>[Apache License 2.0]) <https://code.google.com/p/cld2/>[Compact Language Detector] (CLD) from Google. It is small, fast, ("Compact Language Detector (CLD)") and accurate, and can detect 160+ languages from as little as two sentences. It can even detect multiple languages within a single block of text. Bindings exist for several languages including Python, Perl, JavaScript, PHP, C#/.NET, and R.

Identifying the language of the user's search request is not quite as simple. The CLD is designed for text that is at least 200 characters in length. Shorter amounts of text, such as search keywords, produce much less accurate results. In these cases, it may be preferable to take simple heuristics into account such as the country of origin, the user's selected language, and the HTTP `accept-language` headers.



[[one-lang-docs]] === One Language per Document

A single predominant language per document ((("languages", "one language per document"))(("indices", "documents in different languages"))) requires a relatively simple setup. Documents from different languages can be stored in separate indices—`blogs-en` , `blogs-fr` , and so forth—that use the same type and the same fields for each index, just with different analyzers:

[source,js]

```
PUT /blogs-en { "mappings": { "post": { "properties": { "title": { "type": "string", <1> "fields": { "stemmed": { "type": "string", "analyzer": "english" <2> } }}}}}}
```

```
PUT /blogs-fr { "mappings": { "post": { "properties": { "title": { "type": "string", <1> "fields": { "stemmed": { "type": "string", "analyzer": "french" <2> } }}}}}}
```

```
}}}}}}
```

<1> Both `blogs-en` and `blogs-fr` have a type called `post` that contains the field `title` .

<2> The `title.stemmed` subfield uses a language-specific analyzer.

This approach is clean and flexible. New languages are easy to add--just create a new index--and because each language is completely separate, we don't suffer from the term-frequency and stemming problems described in <>.

The documents of a single language can be queried independently, or queries can target multiple languages by querying multiple indices. We can even specify a preference((("indices_boost parameter", "specifying preference for a specific language"))) for particular languages with the `indices_boost` parameter:

[source,js]

```
GET /blogs-*/post/_search <1> { "query": { "multi_match": { "query": "deja vu", "fields": [ "title", "title.stemmed" ] <2> "type": "most_fields" } }, "indices_boost": { <3> "blogs-en": 3, "blogs-fr": 2 } }
```

```
}
```

<1> This search is performed on any index beginning with `blogs-` .



<2> The `title.stemmed` fields are queried using the analyzer specified in each index.

<3> Perhaps the user's `accept-language` headers showed a preference for English, and then French, so we boost results from each index accordingly. Any other languages will have a neutral boost of `1`.

==== Foreign Words

Of course, these documents may contain words or sentences in other languages, and these words are unlikely to be stemmed correctly. With predominant-language documents, this is not usually a major problem. The user will often search for the exact words--for instance, of a quotation from another language--rather than for inflections of a word. Recall can be improved by using techniques explained in <>.

Perhaps some words like place names should be queryable in the predominant language and in the original language, such as *Munich* and *München*. These words are effectively synonyms, which we discuss in <>.

.Don't Use Types for Languages

You may be tempted to use a separate type for each language,(((`"types"`, "not using for languages")))((`"languages"`, "not using types for"))) instead of a separate index. For best results, you should avoid using types for this purpose. As explained in <>, fields from different types but with the same field name are indexed into the *same inverted index*. This means that the term frequencies from each type (and thus each language) are mixed together.

To ensure that the term frequencies of one language don't pollute those of another, either use a separate index for each language, or a separate field, as explained in the next section.



[[one-lang-fields]] === One Language per Field

For documents that represent entities like products, movies, or legal notices, it is common((("fields", "one language per field")))((("languages", "one language per field"))) for the same text to be translated into several languages. Although each translation could be represented in a single document in an index per language, another reasonable approach is to keep all translations in the same document:

[source,js]

```
{ "title": "Fight club", "title_br": "Clube de Luta", "title_cz": "Klub rvácu", "title_en": "Fight club",  
"title_es": "El club de la lucha", ...
```

}

Each translation is stored in a separate field, which is analyzed according to the language it contains:

[source,js]

```
PUT /movies { "mappings": { "movie": { "properties": { "title": { <1> "type": "string" }, "title_br":  
<2> "type": "string", "analyzer": "brazilian" }, "title_cz": { <2> "type": "string", "analyzer":  
"czech" }, "title_en": { <2> "type": "string", "analyzer": "english" }, "title_es": { <2> "type":  
"string", "analyzer": "spanish" } } } }
```

}

<1> The `title` field contains the original title and uses the `standard` analyzer.

<2> Each of the other fields uses the appropriate analyzer for that language.

Like the *index-per-language* approach, the *field-per-language* approach maintains clean term frequencies. It is not quite as flexible as having separate indices. Although it is easy to add a new field by using the `<>`, those new fields may require new custom analyzers, which can only be set up at index creation time. As a workaround, you can

<http://bit.ly/1B6s0WY>[close] the index, add the new analyzers with the

<http://bit.ly/1zijFPx>[`update-settings` API], then reopen the index, but closing the index

means that it will require some downtime.



The documents of a(("boosting", "query-time", "boosting a field"))) single language can be queried independently, or queries can target multiple languages by querying multiple fields. We can even specify a preference for particular languages by boosting that field:

[source,js]

```
GET /movies/movie/_search { "query": { "multi_match": { "query": "club de la lucha", "fields": [ "title*", "title_es^2" ], <1> "type": "most_fields" } } }
```

```
}
```

<1> This search queries any field beginning with `title` but boosts the `title_es` field by 2. All other fields have a neutral boost of 1.



[[mixed-lang-fields]] === Mixed-Language Fields

Usually, documents that mix multiple languages in a single field come from sources beyond your control, such as`((("languages", "mixed language fields")))((("fields", "mixed language")))` pages scraped from the Web:

[source,js]

```
{ "body": "Page not found / Seite nicht gefunden / Page non trouvée" }
```

They are the most difficult type of multilingual document to handle correctly. Although you can simply use the `standard` analyzer on all fields, your documents will be less searchable than if you had used an appropriate stemmer. But of course, you can't choose just one stemmer--stemmers are language specific. Or rather, stemmers are language and script specific. As discussed in <>, if every language uses a different script, then stemmers can be combined.

Assuming that your mix of languages uses the same script such as Latin, you have three choices available to you:

- Split into separate fields
- Analyze multiple times
- Use n-grams

===== Split into Separate Fields

The Compact Language Detector`((("languages", "mixed language fields", "splitting into separate fields")))((("Compact Language Detector (CLD)")))` mentioned in <> can tell you which parts of the document are in which language. You can split up the text based on language and use the same approach as was used in <>.

===== Analyze Multiple Times

If you primarily deal with a limited number of languages,`((("languages", "mixed language fields", "analyzing multiple times")))((("analyzers", "for mixed language fields")))((("multifields", "analyzing mixed language fields")))`you could use multi-fields to analyze the text once per language:

[source,js]



```
PUT /movies { "mappings": { "title": { "properties": { "title": { <1> "type": "string", "fields": { "de": { <2> "type": "string", "analyzer": "german" }, "en": { <2> "type": "string", "analyzer": "english" }, "fr": { <2> "type": "string", "analyzer": "french" }, "es": { <2> "type": "string", "analyzer": "spanish" } } } } }}
```

```
}
```

<1> The main `title` field uses the `standard` analyzer.

<2> Each subfield applies a different language analyzer to the text in the `title` field.

==== Use n-grams

You could index all words as n-grams, using the (((("n-grams", "for mixed language fields")))) (((("languages", "mixed language fields", "n-grams, indexing words as")))) same approach as described in <>. Most inflections involve adding a suffix (or in some languages, a prefix) to a word, so by breaking each word into n-grams, you have a good chance of matching words that are similar but not exactly the same. This can be combined with the *analyze-multiple times* approach to provide a catchall field for unsupported languages:

[source,js]

```
PUT /movies { "settings": { "analysis": { ... } <1> }, "mappings": { "title": { "properties": { "title": { "type": "string", "fields": { "de": { "type": "string", "analyzer": "german" }, "en": { "type": "string", "analyzer": "english" }, "fr": { "type": "string", "analyzer": "french" }, "es": { "type": "string", "analyzer": "spanish" } } } } } } }
```

```
}
```

<1> In the `analysis` section, we define the same `trigrams` analyzer as described in <>.

<2> The `title.general` field uses the `trigrams` analyzer to index any language.

When querying the catchall `general` field, you can use `minimum_should_match` to reduce the number of low-quality matches. It may also be necessary to boost the other fields slightly more than the `general` field, so that matches on the the main language fields are given more weight than those on the `general` field:

[source,js]



```
GET /movies/movie/_search { "query": { "multi_match": { "query": "club de la lucha", "fields": [ "title^1.5", "title.general" ], <1> "type": "most_fields", "minimum_should_match": "75%" <2> } }
```

}

<1> All `title` or `title.*` fields are given a slight boost over the `title.general` field.

<2> The `minimum_should_match` parameter reduces the number of low-quality matches returned, especially important for the `title.general` field.



==== Conclusion

This chapter has explained how to get started with language analysis using the tools that are available to you out of the box. You may never need more -- it depends on the languages that you have to deal with and your business requirements. Getting language search right is a complex job which often requires custom configuration. The next chapters delve more deeply into the building blocks provided by Elasticsearch that you can join together to build your optimal solution.



[[identifying-words]] == Identifying Words

A word in English is relatively simple to spot: words are separated by whitespace or (some) punctuation.(((“languages”, “identifyig words”)))(((“words”, “identifying”))) Even in English, though, there can be controversy: is *you’re* one word or two? What about *o’clock*, *cooperate*, *half-baked*, or *eyewitness*?

Languages like German or Dutch combine individual words to create longer compound words like *Weißenkopfseeadler* (white-headed sea eagle), but in order to be able to return `Weißenkopfseeadler` as a result for the query `Adler` (eagle), we need to understand how to break up compound words into their constituent parts.

Asian languages are even more complex: some have no whitespace between words, sentences, or even paragraphs.(((“Asian languages”, “identifying words”))) Some words can be represented by a single character, but the same single character, when placed next to other characters, can form just one part of a longer word with a quite different meaning.

It should be obvious that there is no silver-bullet analyzer that will miraculously deal with all human languages. Elasticsearch ships with dedicated analyzers for many languages, and more language-specific analyzers are available as plug-ins.

However, not all languages have dedicated analyzers, and sometimes you won’t even be sure which language(s) you are dealing with. For these situations, we need good standard tools that do a reasonable job regardless of language.



[[standard-analyzer]] === standard Analyzer

The `standard` analyzer is used by default for any full-text `analyzed` string field. (>("standard analyzer")) If we were to reimplement the `standard` analyzer as a <>, it would be defined as follows:

```
[role="pagebreak-before"]
```

[source,js]

```
{ "type": "custom", "tokenizer": "standard", "filter": [ "lowercase", "stop" ] }
```

```
}
```

In <> and <>, we talk about the `lowercase`, and `stop` *token filters*, but for the moment, let's focus on the `standard` *tokenizer*.



[[standard-tokenizer]] === standard Tokenizer

A *tokenizer* accepts a string as input, processes(((“words”, “identifying”, “using standard tokenizer”))))((“standard tokenizer”))))((“tokenizers”))) the string to break it into individual words, or *tokens* (perhaps discarding some characters like punctuation), and emits a *token stream* as output.

What is interesting is the algorithm that is used to *identify* words. The `whitespace` *tokenizer* (((“whitespace tokenizer”))) simply breaks on whitespace--spaces, tabs, line feeds, and so forth--and assumes that contiguous nonwhitespace characters form a single token. For instance:

[source,js]

```
GET /_analyze?tokenizer=whitespace
```

You're the 1st runner home!

This request would return the following terms: You're , the , 1st , runner , home!

The `letter` *tokenizer*, on the other hand, breaks on any character that is not a letter, and so would (((“letter tokenizer”))) return the following terms: You , re , the , st , runner , home .

The `standard` *tokenizer*((“Unicode Text Segmentation algorithm”))) uses the Unicode Text Segmentation algorithm (as defined in <http://unicode.org/reports/tr29/>[Unicode Standard Annex #29]) to find the boundaries *between* words,(((“word boundaries”))) and emits everything in-between. Its knowledge of Unicode allows it to successfully tokenize text containing a mixture of languages.

Punctuation may(((“punctuation”, “in words”))) or may not be considered part of a word, depending on where it appears:

[source,js]

```
GET /_analyze?tokenizer=standard
```

You're my 'favorite'.

In this example, the apostrophe in `You're` is treated as part of the word, while the single quotes in `'favorite'` are not, resulting in the following terms: `You're` , `my` , `favorite` .

[TIP]

The `uax_url_email` tokenizer works`((("uax_url_email tokenizer")))` in exactly the same way as the `standard` tokenizer, except that it recognizes`((("email addresses and URLs, tokenizer for")))` email addresses and URLs and emits them as single tokens. The `standard` tokenizer, on the other hand, would try to break them into individual words. For instance, the email address `joe-bloggs@foo-bar.com` would result in the tokens `joe` , `bloggs` , `foo` , `bar.com` .

=====

The `standard` tokenizer is a reasonable starting point for tokenizing most languages, especially Western languages. In fact, it forms the basis of most of the language-specific analyzers like the `english` , `french` , and `spanish` analyzers. Its support for Asian languages, however, is limited, and you should consider using the `icu_tokenizer` instead, `((("icu_tokenizer")))` which is available in the ICU plug-in.



[[icu-plugin]] === Installing the ICU Plug-in

The [ICU analysis plug-in](https://github.com/elasticsearch/elasticsearch-analysis-icu) for Elasticsearch uses the *International Components for Unicode* (ICU) libraries (see [site.project.org](http://site.icu-project.org)) to provide a rich set of tools for dealing with Unicode. These include the `icu_tokenizer`, which is particularly useful for Asian languages, and a number of token filters that are essential for correct matching and sorting in all languages other than English.

[NOTE]

The ICU plug-in is an essential tool for dealing with languages other than English, and it is highly recommended that you install and use it. Unfortunately, because it is based on the external ICU libraries, different versions of the ICU plug-in may not be compatible with previous versions. When upgrading, you may need to reindex your data.

=====

To install the plug-in, first shut down your Elasticsearch node and then run the following command from the Elasticsearch home directory:

[source,sh]

./bin/plugin -install.elasticsearch/elasticsearch-analysis-icu/\$VERSION <1>

<1> The current `$VERSION` can be found at <https://github.com/elasticsearch/elasticsearch-analysis-icu>.

Once installed, restart Elasticsearch, and you should see a line similar to the following in the startup logs:

```
[INFO][plugins] [Mysterio] loaded [marvel, analysis-icu], sites [marvel]
```

If you are running a cluster with multiple nodes, you will need to install the plug-in on every node in the cluster.



[[icu-tokenizer]] === icu_tokenizer

The `icu_tokenizer` uses the same Unicode Text Segmentation algorithm as the `standard_tokenizer`, but adds better support for some Asian languages by using a dictionary-based approach to identify words in Thai, Lao, Chinese, Japanese, and Korean, and using custom rules to break Myanmar and Khmer text into syllables.

For instance, compare the tokens ((("standard tokenizer", "icu_tokenizer versus"))) produced by the `standard` and `icu_tokenizers`, respectively, when tokenizing ``Hello. I am from Bangkok.'' in Thai:

[source,js]

GET /_analyze?tokenizer=standard

ສ ວ ສ ເ ດ ພ ມ ມ ອ ຈ ກ ກ ວ ຢ ໃ ເ ທ ພ

The standard tokenizer produces two tokens, one for each sentence: ສາສັດໃໝ່, ພມມາ
ຈາກອານຸເທພາະ . That is useful only if you want to search for the whole sentence I am
from Bangkok.''. but not if you want to search for just Bangkok."

source.js

GET /_analyze?tokenizer=icu tokenizer

ສ ວ ສ ເ ດ ພ ມ ມ ອ ຈ ກ ກ ຮ ສ ແ ເ ທ ພ

The `icu_tokenizer`, on the other hand, is able to break up the text into the individual words (ສັງສົດ ແລ້ວ ພມ, ມາ, ຈາກ, ໂຮງ ຖທພາ), making them easier to search.

In contrast, the standard tokenizer ``over-tokenizes'' Chinese and Japanese text, often breaking up whole words into single characters. Because there are no spaces between words, it can be difficult to tell whether consecutive characters are separate words or form a single word. For instance:

- 向 means *facing*, 日 means *sun*, and 葵 means *hollyhock*. When written together, 向日葵 means *sunflower*.



- 五 means *five* or *fifth*, 月 means *month*, and 雨 means *rain*. The first two characters written together as 五月 mean *the month of May*, and adding the third character, 五月雨 means *continuous rain*. When combined with a fourth character, 式, meaning *style*, the word 五月雨式 becomes an adjective for anything consecutive or unrelenting.

Although each character may be a word in its own right, tokens are more meaningful when they retain the bigger original concept instead of just the component parts:

[source.js]

```
GET /_analyze?tokenizer=standard 向日葵
```

```
GET /_analyze?tokenizer=icu_tokenizer
```

向日葵

The `standard` tokenizer in the preceding example would emit each character as a separate token: 向, 日, 葵. The `icu_tokenizer` would emit the single token 向日葵 (sunflower).

Another difference between the `standard` tokenizer and the `icu_tokenizer` is that the latter will break a word containing characters written in different scripts (for example, βeta) into separate tokens—β, eta—while the former will emit the word as a single token: βeta.



[[char-filters]] === Tidying Up Input Text

Tokenizers produce the best results when the input text is clean, valid text, where *valid* means that it follows the punctuation rules that the Unicode algorithm expects.(((("text", "tidying up text input for tokenizers"))))((("words", "identifying", "tidying up text input")))) Quite often, though, the text we need to process is anything but clean. Cleaning it up before tokenization improves the quality of the output.

==== Tokenizing HTML

Passing HTML through the `standard` tokenizer or the `icu_tokenizer` produces poor results.(((("HTML, tokenizing")))) These tokenizers just don't know what to do with the HTML tags. For example:

[source,js]

```
GET /_analyzer?tokenizer=standard
```

Some déjà vu website

The `standard` tokenizer(((("standard tokenizer", "tokenizing HTML")))) confuses HTML tags and entities, and emits the following tokens: `p` , `Some` , `d` , `eacute` , `j` , `grave` , `vu` , `a` , `href` , `http` , `somedomain.com` , `website` , `a` . Clearly not what was intended!

Character filters can be added to an analyzer to (((("character filters")))) preprocess the text before it is passed to the tokenizer. In this case, we can use the `html_strip` character filter(((("analyzers", "adding character filters to"))))((("html_strip character filter")))) to remove HTML tags and to decode HTML entities such as `é` into the corresponding Unicode characters.

Character filters can be tested out via the `analyze` API by specifying them in the query string:

[source,js]

```
GET /_analyzer?tokenizer=standard&char_filters=html_strip
```

Some déjà vu website

To use them as part of the analyzer, they should be added to a `custom` analyzer definition:



[source,js]

```
PUT /my_index { "settings": { "analysis": { "analyzer": { "my_html_analyzer": { "tokenizer": "standard", "char_filter": [ "html_strip" ] } } } }
```

```
}
```

Once created, our new `my_html_analyzer` can be tested with the `analyze` API:

[source,js]

```
GET /my_index/_analyzer?analyzer=my_html_analyzer
```

Some déjà vu website

This emits the tokens that we expect: `Some`, `++déjà++`, `vu`, `website`.

==== Tidying Up Punctuation

The `standard` tokenizer and `icu_tokenizer` both understand that an apostrophe *within* a word should be treated as part of the word, while single quotes that *surround* a word should not.(((("standard tokenizer", "handling of punctuation")))((("icu_tokenizer", "handling of punctuation")))((("punctuation", "tokenizers' handling of")))) Tokenizing the text `You're my favorite`. would correctly emit the tokens `You're, my, favorite`.

Unfortunately,(((("apostrophes"))) Unicode lists a few characters that are sometimes used as apostrophes:

`U+0027` :: Apostrophe (')—the original ASCII character

`U+2018` :: Left single-quotation mark (‘)—opening quote when single-quoting

`U+2019` :: Right single-quotation mark (’)—closing quote when single-quoting, but also the preferred character to use as an apostrophe

Both tokenizers treat these three characters as an apostrophe (and thus as part of the word) when they appear within a word. Then there are another three apostrophe-like characters:

`U+201B` :: Single high-reversed-9 quotation mark (‘)—same as `U+2018` but differs in appearance

`U+0091` :: Left single-quotation mark in ISO-8859-1—should not be used in Unicode



U+0092 :: Right single-quotation mark in ISO-8859-1—should not be used in Unicode

Both tokenizers treat these three characters as word boundaries--a place to break text into tokens.(((“quotation marks”))) Unfortunately, some publishers use U+201B as a stylized way to write names like M'coy , and the second two characters may well be produced by your word processor, depending on its age.

Even when using the ‘acceptable’ quotation marks, a word written with a single right quotation mark— You’re —is not the same as the word written with an apostrophe— You're`—which means that a query for one variant will not find the other.

Fortunately, it is possible to sort out this mess with the mapping character filter,((("character filters", "mapping character filter")))((("mapping character filter"))) which allows us to replace all instances of one character with another. In this case, we will replace all apostrophe variants with the simple U+0027 apostrophe:

[source,js]

```
PUT /my_index { "settings": { "analysis": { "char_filter": { <1> "quotes": { "type": "mapping", "mappings": [ <2> "\u0091=>\u0027", "\u0092=>\u0027", "\u2018=>\u0027", "\u2019=>\u0027", "\u201b=>\u0027" ] } }, "analyzer": { "quotes_analyzer": { "tokenizer": "standard", "char_filter": [ "quotes" ] <3> } } } }
```

}

<1> We define a custom char_filter called quotes that maps all apostrophe variants to a simple apostrophe.

<2> For clarity, we have used the JSON Unicode escape syntax for each character, but we could just have used the characters themselves: “ ‘=> ‘ ” .

<3> We use our custom quotes character filter to create a new analyzer called quotes_analyzer .

As always, we test the analyzer after creating it:

[source,js]

```
GET /my_index/_analyze?analyzer=quotes_analyzer
```



You're my 'favorite' M'Coy

This example returns the following tokens, with all of the in-word quotation marks replaced by apostrophes: You're , my , favorite , M'Coy .

The more effort that you put into ensuring that the tokenizer receives good-quality input, the better your search results will be.



[[token-normalization]] == Normalizing Tokens

Breaking text into tokens is ((("normalization", "of tokens")))((("tokens", "normalizing")))only half the job. To make those tokens more easily searchable, they need to go through a *normalization* process to remove insignificant differences between otherwise identical words, such as uppercase versus lowercase. Perhaps we also need to remove significant differences, to make `esta` , `ésta` , and `está` all searchable as the same word. Would you search for `déjà vu` , or just for `deja vu` ?

This is the job of the token filters, which((("token filters")))) receive a stream of tokens from the tokenizer. You can have multiple token filters, each doing its particular job. Each receives the new token stream as output by the token filter before it.



[[lowercase-token-filter]] === In That Case

The most frequently used token filter is the `lowercase` filter, which does exactly what you would expect; it transforms (((`"tokens"`, `"normalizing"`, `"lowercase filter"`)))((`"lowercase token filter"`)))each token into its lowercase form:

[source,js]

```
GET /_analyze?tokenizer=standard&filters=lowercase
```

The QUICK Brown FOX! <1>

<1> Emits tokens `the` , `quick` , `brown` , `fox`

It doesn't matter whether users search for `fox` or `Fox` , as long as the same analysis process is applied at query time and at search time. The `lowercase` filter will transform a query for `Fox` into a query for `fox` , which is the same token that we have stored in our inverted index.

To use token filters as part of the analysis process, we (((`"analyzers"`, `"using token filters"`)))((`"token filters"`, `"using with analyzers"`)))can create a `custom` analyzer:

[source,js]

```
PUT /my_index { "settings": { "analysis": { "analyzer": { "my_lowecaser": { "tokenizer": "standard", "filter": [ "lowercase" ] } } } }
```

```
}
```

And we can test it out with the `analyze` API:

[source,js]

```
GET /my_index/_analyze?analyzer=my_lowecaser
```

The QUICK Brown FOX! <1>



<1> Emits tokens the , quick , brown , fox



[[asciifolding-token-filter]] === You Have an Accent

English uses diacritics (like ‘́’, ‘^’, and ‘”’) only for imported words--like rôle, ++déjà++, and däis—but usually they are optional. (((“diacritics”)))((“tokens”, “normalizing”, “diacritics”))) Other languages require diacritics in order to be correct. Of course, just because words are spelled correctly in your index doesn’t mean that the user will search for the correct spelling.

It is often useful to strip diacritics from words, allowing rôle to match role, and vice versa. With Western languages, this can be done with the asciifolding character filter. (((“asciifolding character filter”))) Actually, it does more than just strip diacritics. It tries to convert many Unicode characters into a simpler ASCII representation:

- ß => ss
- æ => ae
- þ => l
- ū => m
- ?? => ??
- ② => 2
- ⁶ => 6

Like the lowercase filter, the asciifolding filter doesn’t require any configuration but can be included directly in a custom analyzer:

[source,js]

```
PUT /my_index { "settings": { "analysis": { "analyzer": { "folding": { "tokenizer": "standard", "filter": [ "lowercase", "asciifolding" ] } } } } }
```

```
GET /my_index?analyzer=folding
```

My œsophagus caused a déâcle <1>

<1> Emits my, œsophagus, caused, a, déâcle

==== Retaining Meaning

Of course, when you strip diacritical marks from a word, you lose meaning. For instance, consider(((“diacritics”, “stripping, meaning loss from”))) these three (((“Spanish”, “stripping diacritics, meaning loss from”))) Spanish words:

esta ::

Feminine form of the adjective *this*, as in *esta silla* (this chair) or *esta* (this one).



esta ::

An archaic form of `esta`.

`está` ::

The third-person form of the verb *estar* (to be), as in `está feliz` (he is happy).

While we would like to conflate the first two forms, they differ in meaning from the third form, which we would like to keep separate. Similarly:

`sé` ::

The first person form of the verb *saber* (to know) as in `Yo sé` (I know).

`se` ::

The third-person reflexive pronoun used with many verbs, such as `se sabe` (it is known).

Unfortunately, there is no easy way to separate words that should have their diacritics removed from words that shouldn't. And it is quite likely that your users won't know either.

Instead, we index the text twice: once in the original form and once with diacritics
((("indexing", "text with diacritics removed"))))removed:

[source,js]

```
PUT /my_index/_mapping/my_type { "properties": { "title": { <1> "type": "string", "analyzer": "standard", "fields": { "folded": { <2> "type": "string", "analyzer": "folding" } } } }}
```

}

<1> The `title` field uses the `standard` analyzer and will contain the original word with diacritics in place.

<2> The `title.folded` field uses the `folding` analyzer, which strips the diacritical marks.
((("folding analyzer"))))

You can test the field mappings by using the `analyze` API on the sentence *Esta está loca* (This woman is crazy):

[source,js]

```
GET /my_index/_analyze?field=title <1> Esta está loca
```

```
GET /my_index/_analyze?field=title.folded <2>
```



Esta está loca

<1> Emits `esta`, `está`, `loca`

<2> Emits `esta`, `esta`, `loca`

Let's index some documents to test it out:

[source,js]

```
PUT /my_index/my_type/1 { "title": "Esta loca!" }
```

```
PUT /my_index/my_type/2
```

```
{ "title": "Está loca!" }
```

Now we can search across both fields, using the `multi_match` query in <> to combine the scores from each field:

[source,js]

```
GET /my_index/_search { "query": { "multi_match": { "type": "most_fields", "query": "esta loca", "fields": [ "title", "title.folded" ] } } }
```

```
}
```

Running this query through the `validate-query` API helps to explain how the query is executed:

[source,js]

```
GET /my_index/_validate/query?explain { "query": { "multi_match": { "type": "most_fields", "query": "está loca", "fields": [ "title", "title.folded" ] } } }
```

```
}
```



The `multi-match` query searches for the original form of the word (`está`) in the `title` field, and the form without diacritics `esta` in the `title.folded` field:

```
(title:está      title:loca      )
(title.folded:esta title.folded:loca)
```

It doesn't matter whether the user searches for `esta` or `está`; both documents will match because the form without diacritics exists in the `title.folded` field. However, only the original form exists in the `title` field. This extra match will push the document containing the original form of the word to the top of the results list.

We use the `title.folded` field to *widen the net* in order to match more documents, and use the original `title` field to push the most relevant document to the top. This same technique can be used wherever an analyzer is used, to increase matches at the expense of meaning.

[TIP]

The `asciifolding` filter does have an option called `preserve_original` that allows you to index the(("asciifolding character filter", "preserve_original option")) original token and the folded token in the same position in the same field. With this option enabled, you would end up with something like this:

Position 1	Position 2
-----	-----
(ésta,esta)	loca
-----	-----

While this appears to be a nice way to save space, it does mean that you have no way of saying, ``Give me an exact match on the original word.'' Mixing tokens with and without diacritics can also end up interfering with term-frequency counts, resulting in less-reliable relevance calculations.

As a rule, it is cleaner to index each field variant into a separate field, as we have done in this section.



[[unicode-normalization]] === Living in a Unicode World

When Elasticsearch compares one token with another, it does so at the byte level.

((("Unicode", "token normalization and")))((("tokens", "normalizing", "Unicode and"))))In other words, for two tokens to be considered the same, they need to consist of exactly the same bytes. Unicode, however, allows you to write the same letter in different ways.

For instance, what's the difference between é and é? It depends on who you ask. According to Elasticsearch, the first one consists of the two bytes `0xC3 0xA9`, and the second one consists of three bytes, `0x65 0xCC 0x81`.

According to Unicode, the differences in how they are represented as bytes is irrelevant, and they are the same letter. The first one is the single letter `é`, while the second is a plain `e` combined with an acute accent `' + ' + ' + '`.

If you get your data from more than one source, it may happen that you have the same letters encoded in different ways, which may result in one form of `++déjà++` not matching another!

Fortunately, a solution is at hand. There are four Unicode *normalization forms*, all of which convert Unicode characters into a standard format, making all characters((("Unicode", "normalization forms")))) comparable at a byte level: `nfc`, `nfd`, `nfkc`, `nfkd`.((("nfkd normalization form"))))((("nfkc normalization form"))))((("nfd normalization form"))))((("nfc normalization form"))))

.Unicode Normalization Forms

The *composed* forms—`nfc` and `nfkc`—represent characters in the fewest bytes possible.((("composed forms (Unicode normalization)")))) So `é` is represented as the single letter `é`. The *decomposed* forms—`nfd` and `nfkd`—represent characters by their constituent parts, that is `e + ' + ' + '`.((("decomposed forms (Unicode normalization)"))))

The *canonical* forms—`nfc` and `nfd`—represent ligatures like `ffi` or `œ` as a single character,((("canonical forms (Unicode normalization)")))) while the *compatibility* forms—`nfkc` and `nfkd`—break down these composed characters into a simpler multiletter equivalent: `f + f + i` or `o + e`.

It doesn't really matter which normalization form you choose, as long as all your text is in the same form. That way, the same tokens consist of the same bytes. That said, the *compatibility* forms ((("compatibility forms (Unicode normalization)"))))allow you to compare ligatures like `ffi` with their simpler representation, `ffii`.



You can use the `icu_normalizer` token filter to (((`"icu_normalizer token filter"`)))ensure that all of your tokens are in the same form:

[source,js]

```
PUT /my_index { "settings": { "analysis": { "filter": { "nfkc_normalizer": { <1> "type": "icu_normalizer", "name": "nfkc" } }, "analyzer": { "my_normalizer": { "tokenizer": "icu_tokenizer", "filter": [ "nfkc_normalizer" ] } } } }
```

}

<1> Normalize all tokens into the `nfkc` normalization form.

[TIP]

Besides the `icu_normalizer` token filter mentioned previously, there is also an `icu_normalizer character` filter, which((("icu_normalizer character filter"))) does the same job as the token filter, but does so before the text reaches the tokenizer. When using the `standard` tokenizer or `icu_tokenizer`, this doesn't really matter. These tokenizers know how to deal with all forms of Unicode correctly.

However, if you plan on using a different tokenizer, such as the `ngram`, `edge_ngram`, or `pattern` tokenizers, it would make sense to use the `icu_normalizer` character filter in preference to the token filter.

=====

Usually, though, you will want to not only normalize the byte order of tokens, but also lowercase them. This can be done with `icu_normalizer`, using the custom normalization form `nfkc_cf`, which we discuss in the next section.



[[case-folding]] === Unicode Case Folding

Humans are nothing if not inventive,((("tokens", "normalizing", "Unicode case folding")))((("Unicode", "case folding")))) and human language reflects that. Changing the case of a word seems like such a simple task, until you have to deal with multiple languages.

Take, for example, the lowercase German letter `\u00f6`. Converting that to upper case gives you `\u00d6`, which converted back to lowercase gives you `\u00f6`. Or consider the Greek letter `\u03c6` (sigma, when used at the end of a word). Converting it to uppercase results in `\u0393`, which converted back to lowercase, gives you `\u03c6`.

The whole point of lowercasing terms is to make them *more* likely to match, not less! In Unicode, this job is done by case folding rather((("case folding")))) than by lowercasing. *Case folding* is the act of converting words into a (usually lowercase) form that does not necessarily result in the correct spelling, but does allow case-insensitive comparisons.

For instance, the letter `\u00f6`, which is already lowercase, is *folded* to `\u00d6`. Similarly, the lowercase `\u03c6` is folded to `\u0393`, to make `\u00d6`, `\u03c6`, and `\u0393` comparable, no matter where the letter appears in a word.((("nfkc_cf normalization form")))((("icu_normalizer token filter", "nfkc_cf normalization form"))))

The default normalization form that the `icu_normalizer` token filter uses is `nfkc_cf`. Like the `nfkc` form, this does the following:

- Composes characters into the shortest byte representation
- Uses compatibility mode to convert characters like `\u00e1` into the simpler `\u00e1`

But it also does this:

- Case-folds characters into a form suitable for case comparison

In other words, `nfkc_cf` is the equivalent of the `lowercase` token filter, but suitable for use with all languages.((("lowercase token filter", "nfkccf normalization form and")))) *The_on-steroids* equivalent of the `standard` analyzer would be the following:

[source,js]

```
PUT /my_index { "settings": { "analysis": { "analyzer": { "my_lowercaser": { "tokenizer": "icu_tokenizer", "filter": [ "icu_normalizer" ] <1> } } } }
```

```
}
```

<1> The `icu_normalizer` defaults to the `nfkc_cf` form.



We can compare the results of running `Weīkopfseeadler` and `WEISSKOPFSEEADLER` (the uppercase equivalent) through the `standard` analyzer and through our Unicode-aware analyzer:

[source,js]

```
GET /_analyze?analyzer=standard <1> Weīkopfseeadler WEISSKOPFSEEADLER
```

```
GET /my_index/_analyze?analyzer=my_lowercaser <2>
```

Weīkopfseeadler WEISSKOPFSEEADLER

<1> Emits tokens `weīkopfseeadler` , `weisskopfseeadler`

<2> Emits tokens `weisskopfseeadler` , `weisskopfseeadler`

The `standard` analyzer emits two different, incomparable tokens, while our custom analyzer produces tokens that are comparable, regardless of the original case.



[[character-folding]] === Unicode Character Folding

In the same way as the `lowercase` token filter is a good starting point for many languages(((("Unicode", "character folding")))((("tokens", "normalizing", "Unicode character folding")))) but falls short when exposed to the entire tower of Babel, so the <> requires a more effective Unicode *character-folding* counterpart(((("character folding")))) for dealing with the many languages of the world.((("asciifolding token filter")))

The `icu_folding` token filter (provided by the <>) does the same job as the `asciifolding` filter, (((("icu_folding token filter"))))but extends the transformation to scripts that are not ASCII-based, such as Greek, Hebrew, Han, conversion of numbers in other scripts into their Latin equivalents, plus various other numeric, symbolic, and punctuation transformations.

The `icu_folding` token filter applies Unicode normalization and case folding from `nfkc_cf` automatically,(((("nfkc_cf normalization form")))) so the `icu_normalizer` is not required:

[source,js]

```
PUT /my_index { "settings": { "analysis": { "analyzer": { "my_folder": { "tokenizer": "icu_tokenizer", "filter": [ "icu_folding" ] } } } } }
```

```
GET /my_index/_analyze?analyzer=my_folder
```

١٢٣٤٥ <1>

<1> The Arabic numerals ١٢٣٤٥ are folded to their Latin equivalent: 12345 .

If there are particular characters that you would like to protect from folding, you can use a http://icu-project.org/apiref/icu4j/com/ibm/icu/text/UnicodeSet.html#_UnicodeSet (much like a character class in regular expressions) to specify which Unicode characters may be folded. For instance, to exclude the Swedish letters å, ä, ö, ++Å++, Ä, and ö from folding, you would specify a character class representing all Unicode characters, except for those letters: [^åäöÅÄÖ] (^ means *everything except*).((("swedish_folding filter")))((("swedish analyzer")))

[source,js]

```
PUT /my_index { "settings": { "analysis": { "filter": { "swedish_folding": { <1> "type": "icu_folding", "unicodeSetFilter": "åäöÅÄÖ" } }, "analyzer": { "swedish_analyzer": { <2> "tokenizer": "icu_tokenizer", "filter": [ "swedish_folding", "lowercase" ] } } } } }
```



<1> The `swedish_folding` token filter customizes the `icu_folding` token filter to exclude Swedish letters, both uppercase and lowercase.

<2> The `swedish` analyzer first tokenizes words, then folds each token by using the `swedish_folding` filter, and then lowercases each token in case it includes some of the uppercase excluded letters: ++Å++, Ä, or ö .



[[sorting-collations]] === Sorting and Collations

So far in this chapter, we have looked at how to normalize tokens for the purposes of search. ((("tokens", "normalizing", "for sorting and collation"))) The final use case to consider in this chapter is that of string sorting.((("sorting")))

In <>, we explained that Elasticsearch cannot sort on an `analyzed` string field, and demonstrated how to use `multifields` to index the same field once as an `analyzed` field for search, and once as a `not_analyzed` field for sorting.((("not_analyzed fields", "for string sorting")))((("analyzed fields", "for search")))

The problem with sorting on an `analyzed` field is not that it uses an analyzer, but that the analyzer tokenizes the string value into multiple tokens, like a *bag of words*, and Elasticsearch doesn't know which token to use for sorting.

Relying on a `not_analyzed` field for sorting is inflexible: it allows us to sort on only the exact value of the original string. However, we *can* use analyzers to achieve other sort orders, as long as our chosen analyzer always emits only a single token for each string value.

[[case-insensitive-sorting]] ===== Case-Insensitive Sorting

Imagine that we have three `user` documents whose `name` fields contain `Boffey`, ((("case insensitive sorting")))((("sorting", "case insensitive"))) `BROWN`, and `bailey`, respectively. First we will apply the technique described in <> of using a `not_analyzed` field for sorting:

[source,js]

```
PUT /my_index { "mappings": { "user": { "properties": { "name": { <1> "type": "string", "fields": { "raw": { <2> "type": "string", "index": "not_analyzed" } } } } } }
```

```
}
```

<1> The `analyzed` `name` field is used for search.

<2> The `not_analyzed` `name.raw` field is used for sorting.

We can index some documents and try sorting:

[source,js]

```
PUT /my_index/user/1 { "name": "Boffey" }
```



```
PUT /my_index/user/2 { "name": "BROWN" }
```

```
PUT /my_index/user/3 { "name": "bailey" }
```

GET /my_index/user/_search?sort=name.raw

The preceding search request would return the documents in this order: `BROWN` , `Boffey` , `bailey` . This is known as *lexicographical order* as (((("lexicographical order")))) (((("alphabetical order")))) opposed to *alphabetical order*. Essentially, the bytes used to represent capital letters have a lower value than the bytes used to represent lowercase letters, and so the names are sorted with the lowest bytes first.

That may make sense to a computer, but doesn't make much sense to human beings who would reasonably expect these names to be sorted alphabetically, regardless of case. To achieve this, we need to index each name in a way that the byte ordering corresponds to the sort order that we want.

In other words, we need an analyzer that will emit a single lowercase token:

[source,js]

```
PUT /my_index { "settings": { "analysis": { "analyzer": { "case_insensitive_sort": { "tokenizer": "keyword", <1> "filter": [ "lowercase" ] <2> } } } }
```

```
}
```

<1> The `keyword` tokenizer emits the original input string as a single unchanged token. (((("keyword tokenizer"))))

<2> The `lowercase` token filter lowers the token.

With(((("lowercase token filter")))) the `case_insensitive_sort` analyzer in place, we can now use it in our multifield:

[source,js]

```
PUT /my_index/_mapping/user { "properties": { "name": { "type": "string", "fields": { "lower_case_sort": { <1> "type": "string", "analyzer": "case_insensitive_sort" } } } } }
```

```
PUT /my_index/user/1 { "name": "Boffey" }
```



```
PUT /my_index/user/2 { "name": "BROWN" }
```

```
PUT /my_index/user/3 { "name": "bailey" }
```

GET /my_index/user/_search? sort=name.lower_case_sort

<1> The `name.lower_case_sort` field will provide us with case-insensitive sorting.

The preceding search request returns our documents in the order that we expect: `bailey`, `Boffey`, `BROWN`.

But is this order correct? It appears to be correct as it matches our expectations, but our expectations have probably been influenced by the fact that this book is in English and all of the letters used in our example belong to the English alphabet.

What if we were to add the German name `Böhm`?

Now our names would be returned in this order: `bailey`, `Boffey`, `BROWN`, `Böhm`. The reason that `böhm` comes after `BROWN` is that these words are still being sorted by the values of the bytes used to represent them, and an `r` is stored as the byte `0x72`, while `ö` is stored as `0xF6` and so is sorted last. The byte value of each character is an accident of history.

Clearly, the default sort order is meaningless for anything other than plain English. In fact, there is no ``right'' sort order. It all depends on the language you speak.

==== Differences Between Languages

Every language has its own sort order, and(((("sorting", "differences between languages"))))((("languages", "sort order, differences in")))) sometimes even multiple sort orders. (((("Swedish, sort order"))))((("German", "sort order"))))((("English", "sort order")))) Here are a few examples of how our four names from the previous section would be sorted in different contexts:

- English: `bailey`, `boffey`, `böhm`, `brown`
- German: `bailey`, `boffey`, `böhm`, `brown`
- German phonebook: `bailey`, `böhm`, `boffey`, `brown`
- Swedish: `bailey`, `boffey`, `brown`, `böhm`

[NOTE]



The reason that the German phonebook sort order places `böhm` before `boffey` is that ö and oe are considered synonyms when dealing with names and

places, so böhm is sorted as if it had been written as boehm .

`[[uca]]` ===== Unicode Collation Algorithm

Collation is the process of sorting text into a predefined order.(((("collation"))))((("Unicode Collation Algorithm (UCA")))) The *Unicode Collation Algorithm*, or UCA (see <http://www.unicode.org/reports/tr10/>) defines a method of sorting strings into the order defined in a *Collation Element Table* (usually referred to just as a *collation*).

The UCA also defines the *Default Unicode Collation Element Table*, or *DUCET*, which defines the default sort order(((("Default Unicode Collation Element Table (DUCET")))) for all Unicode characters, regardless of language. As you have already seen, there is no single correct sort order, so DUCET is designed to annoy as few people as possible as seldom as possible, but it is far from being a panacea for all sorting woes.

Instead, language-specific collations(((("languages", "collations")))) exist for pretty much every language under the sun. Most use DUCET as their starting point and add a few custom rules to deal with the peculiarities of each language.

The UCA takes a string and a collation as inputs and outputs a binary sort key. Sorting a collection of strings according to the specified collation then becomes a simple comparison of their binary sort keys.

===== Unicode Sorting

[TIP]

The approach described in this section will probably change in (((("Unicode", "sorting"))))((("sorting", "Unicode"))))a future version of Elasticsearch. Check the <> documentation for the latest information.

=====

The `icu_collation` token filter defaults(((("icu_collation token filter")))) to using the DUCET collation for sorting. This is already an improvement over the default sort. To use it, all we need to do is to create an analyzer that uses the default `icu_collation` filter:



[source,js]

```
PUT /my_index { "settings": { "analysis": { "analyzer": { "ducet_sort": { "tokenizer": "keyword", "filter": [ "icu_collation" ] <1> } } } }
```

```
}
```

<1> Use the default DUCET collation.

Typically, the field that we want to sort on is also a field that we want to search on, so we use the same multifield approach as we used in <>:

[source,js]

```
PUT /my_index/_mapping/user { "properties": { "name": { "type": "string", "fields": { "sort": { "type": "string", "analyzer": "ducet_sort" } } } }
```

```
}
```

With this mapping, the `name.sort` field will contain a sort key that will be used only for sorting. (((Default Unicode Collation Element Table (DUCET))))(((Unicode Collation Algorithm (UCA)))) We haven't specified a language, so it defaults to using the <>.

Now, we can reindex our example docs and test the sorting:

[source,js]

```
PUT /my_index/user/_bulk { "index": { "_id": 1 } } { "name": "Boffey" } { "index": { "_id": 2 } } { "name": "BROWN" } { "index": { "_id": 3 } } { "name": "bailey" } { "index": { "_id": 4 } } { "name": "Böhm" }
```

GET /my_index/user/_search?sort=name.sort

[NOTE]



Note that the `sort` key returned with each document, which in earlier examples looked like `brown` and `böhm`, now looks like gobbledegook: `𢂔𢂔𢂔𢂔\u0001`. The reason is that the `icu_collation` filter emits keys

intended only for efficient sorting, not for any other purposes.

The preceding search returns our docs in this order: `bailey`, `Boffey`, `Böhm`, `BROWN`. This is already an improvement, as the sort order is now correct for English and German, but it is still incorrect for German phonebooks and Swedish. The next step is to customize our mapping for different languages.

==== Specifying a Language

The `icu_collation` filter can be `((("icu_collation token filter", "specifying a language")) (({"languages", "collation table for a specific language, icu_collation filter using"}))` configured to use the collation table for a specific language, a country-specific version of a language, or some other subset such as German phonebooks. This can be done by creating a custom version of the token filter by `((("German", "collation table for, icu_collation filter using")))` using the `language`, `country`, and `variant` parameters as follows:

English:: +

[source,json]

```
{ "language": "en" }
```

German:: +

[source,json]

```
{ "language": "de" }
```

Austrian German:: +

[source,json]



```
{ "language": "de", "country": "AT" }
```

German phonebooks:: +

[source,json]

```
{ "language": "en", "variant": "@collation=phonebook" }
```

[TIP]

You can read more about the locales supported by ICU at: <http://bit.ly/1u9LEdp>.

=====

This example shows how to set up the German phonebook sort order:

[source,js]

```
PUT /my_index { "settings": { "number_of_shards": 1, "analysis": { "filter": { "german_phonebook": { <1> "type": "icu_collation", "language": "de", "country": "DE", "variant": "@collation=phonebook" } }, "analyzer": { "german_phonebook": { <2> "tokenizer": "keyword", "filter": [ "german_phonebook" ] } } }, "mappings": { "user": { "properties": { "name": { "type": "string", "fields": { "sort": { <3> "type": "string", "analyzer": "german_phonebook" } } } } } } }
```

}

<1> First we create a version of the `icu_collation` customized for the German phonebook collation.

<2> Then we wrap that up in a custom analyzer.

<3> And we apply it to our `name.sort` field.

Reindex the data and repeat the same search as we used previously:



[source,js]

```
PUT /my_index/user/_bulk { "index": { "_id": 1 } } { "name": "Boffey" } { "index": { "_id": 2 } } { "name": "BROWN" } { "index": { "_id": 3 } } { "name": "bailey" } { "index": { "_id": 4 } } { "name": "Böhm" }
```

GET /my_index/user/_search?sort=name.sort

This now returns our docs in this order: `bailey` , `Böhm` , `Boffey` , `BROWN` . In the German phonebook collation, `Böhm` is the equivalent of `Boehm` , which comes before `Boffey` .

===== Multiple sort orders

The same field can support multiple ((("sorting", "multiple sort orders supported by same field"))) sort orders by using a multifield for each language:

[source,js]

```
PUT /my_index/_mapping/_user { "properties": { "name": { "type": "string", "fields": { "default": { "type": "string", "analyzer": "ducet" <1> }, "french": { "type": "string", "analyzer": "french" <1> }, "german": { "type": "string", "analyzer": "german_phonebook" <1> }, "swedish": { "type": "string", "analyzer": "swedish" <1> } } } }
```

}

<1> We would need to create the corresponding analyzers for each of these collations.

With this mapping in place, results can be ordered correctly for French, German, and Swedish users, just by sorting on the `name.french` , `name.german` , or `name.swedish` fields. Unsupported languages can fall back to using the `name.default` field, which uses the DUCET sort order.

===== Customizing Collations

The `icu_collation` token filter takes(((("collation", "customizing collations"))))((("icu_collation token filter", "customizing collations")))) many more options than just `language` , `country` , and `variant` , which can be used to tailor the sorting algorithm. Options are available that will do the following:

- Ignore diacritics
- Order uppercase first or last, or ignore case



- Take punctuation and whitespace into account or ignore it
- Sort numbers as strings or by their numeric value
- Customize existing collations or define your own custom collations

Details of these options are beyond the scope of this book, but more information can be found in the <https://github.com/elasticsearch/elasticsearch-analysis-icu>[ICU] plug-in documentation] and in the <http://userguide.icu-project.org/collation/concepts>[ICU project collation documentation].



[[stemming]] == Reducing Words to Their Root Form

Most languages of the world are *inflected*, meaning (((languages", "inflection in))) (((words", "stemming", see="stemming words"))))((("stemming words"))))that words can change their form to express differences in the following:

- *Number*: fox, foxes
- *Tense*: pay, paid, paying
- *Gender*: waiter, waitress
- *Person*: hear, hears
- *Case*: I, me, my
- *Aspect*: ate, eaten
- *Mood*: so be it, were it so

While inflection aids expressivity, it interferes(((inflection))) with retrievability, as a single root *word sense* (or meaning) may be represented by many different sequences of letters. (((English", "inflection in))) English is a weakly inflected language (you could ignore inflections and still get reasonable search results), but some other languages are highly inflected and need extra work in order to achieve high-quality search results.

Stemming attempts to remove the differences between inflected forms of a word, in order to reduce each word to its root form. For instance `foxes` may be reduced to the root `fox`, to remove the difference between singular and plural in the same way that we removed the difference between lowercase and uppercase.

The root form of a word may not even be a real word. The words `jumping` and `jumpiness` may both be stemmed to `jumpi`. It doesn't matter--as long as the same terms are produced at index time and at search time, search will just work.

If stemming were easy, there would be only one implementation. Unfortunately, stemming is an inexact science that (((stemming words", "understemming and overstemming"))))suffers from two issues: understemming and overstemming.

Understemming is the failure to reduce words with the same meaning to the same root. For example, `jumped` and `jumps` may be reduced to `jump`, while `jumping` may be reduced to `jumpi`. Understemming reduces retrieval relevant documents are not returned.

Overstemming is the failure to keep two words with distinct meanings separate. For instance, `general` and `generate` may both be stemmed to `gener`. Overstemming reduces precision: irrelevant documents are returned when they shouldn't be.

.Lemmatization

A *lemma* is the canonical, or dictionary, form ((("lemma"))) of a set of related words--the lemma of `paying`, `paid`, and `pays` is `pay`. Usually the lemma resembles the words it is related to but sometimes it doesn't -- the lemma of `is`, `was`, `am`, and `being` is `be`.

Lemmatization, like stemming, tries to group related words,((("lemmatisation"))) but it goes one step further than stemming in that it tries to group words by their *word sense*, or meaning. The same word may represent two meanings—for example, *wake* can mean *to wake up* or *a funeral*. While lemmatization would try to distinguish these two word senses, stemming would incorrectly conflate them.

Lemmatization is a much more complicated and expensive process that needs to understand the context in which words appear in order to make decisions about what they mean. In practice, stemming appears to be just as effective as lemmatization, but with a much lower cost.

First we will discuss the two classes of stemmers available in Elasticsearch—<> and <>—and then look at how to choose the right stemmer for your needs in <>. Finally, we will discuss options for tailoring stemming in <> and <>.



[[algorithmic-stemmers]] === Algorithmic Stemmers

Most of the stemmers available in Elasticsearch are algorithmic(("stemming words", "algorithmic stemmers")) in that they apply a series of rules to a word in order to reduce it to its root form, such as stripping the final `s` or `es` from plurals. They don't have to know anything about individual words in order to stem them.

These algorithmic stemmers have the advantage that they are available out of the box, are fast, use little memory, and work well for regular words. The downside is that they don't cope well with irregular words like `be`, `are`, and `am`, or `mice` and `mouse`.

One of the earliest stemming algorithms(("English", "stemmers for"))(("Porter stemmer for English")) is the Porter stemmer for English, which is still the recommended English stemmer today. Martin Porter subsequently went on to create the [http://snowball.tartarus.org/\[Snowball language\]](http://snowball.tartarus.org/[Snowball language]) for creating stemming algorithms, and a number(("Snowball langauge (stemmers)")) of the stemmers available in Elasticsearch are written in Snowball.

[TIP]

The [http://bit.ly/1I0bUjZ\[`kstem` token filter\]](http://bit.ly/1I0bUjZ[`kstem` token filter]) is a stemmer for English which(("kstem token filter")) combines the algorithmic approach with a built-in dictionary. The dictionary contains a list of root words and exceptions in order to avoid conflating words incorrectly. `kstem` tends to stem less aggressively than the Porter stemmer.

==== Using an Algorithmic Stemmer

While you(("stemming words", "algorithmic stemmers", "using")) can use the [http://bit.ly/17LseXy\[`porter_stem`\]](http://bit.ly/17LseXy[`porter_stem`]) or [http://bit.ly/1I0bUjZ\[`kstem`\]](http://bit.ly/1I0bUjZ[`kstem`]) token filter directly, or create a language-specific Snowball stemmer with the [http://bit.ly/1Cr4tNI\[`snowball`\]](http://bit.ly/1Cr4tNI[`snowball`]) token filter, all of the algorithmic stemmers are exposed via a single unified interface: the [http://bit.ly/1AUfpDN\[`stemmer` token filter\]](http://bit.ly/1AUfpDN[`stemmer` token filter]), which accepts the `language` parameter.

For instance, perhaps you find the default stemmer used by the `english` analyzer to be too aggressive and(("english analyzer", "default stemmer, examining")) you want to make it less aggressive. The first step is to look up the configuration for the `english` analyzer in the [http://bit.ly/1xtdoJV\[language analyzers\]](http://bit.ly/1xtdoJV[language analyzers]) documentation, which shows the following:

[source,js]



```
{ "settings": { "analysis": { "filter": { "english_stop": { "type": "stop", "stopwords": "_english" }, "english_keywords": { "type": "keyword_marker", <1> "keywords": [] }, "english_stemmer": { "type": "stemmer", "language": "english" <2> }, "english_possessive_stemmer": { "type": "stemmer", "language": "possessive_english" <2> } }, "analyzer": { "english": { "tokenizer": "standard", "filter": [ "english_possessive_stemmer", "lowercase", "english_stop", "english_keywords", "english_stemmer" ] } } }}
```

}

<1> The `keyword_marker` token filter lists words that should not be stemmed.

((("keyword_marker token filter"))) This defaults to the empty list.

<2> The `english` analyzer uses two stemmers: the `possessive_english` and the `english` stemmer. The ((("english stemmer")))((("possessive_english stemmer"))))possessive stemmer removes 's from any words before passing them on to the `english_stop`, `english_keywords`, and `english_stemmer`.

Having reviewed the current configuration, we can use it as the basis for a new analyzer, with((("english analyzer", "customizing the stemmer"))) the following changes:

- Change the `english_stemmer` from `english` (which maps to the [http://bit.ly/17LseXy\['porter_stem'\]](http://bit.ly/17LseXy['porter_stem']) token filter) to `light_english` (which maps to the less aggressive [http://bit.ly/1I0bUjZ\['kstem'\]](http://bit.ly/1I0bUjZ['kstem']) token filter).
- Add the <> token filter to remove any diacritics from foreign words.((("asciifolding token filter")))
- Remove the `keyword_marker` token filter, as we don't need it. (We discuss this in more detail in <>.)

Our new custom analyzer would look like this:

[source,js]

```
PUT /myindex { "settings": { "analysis": { "filter": { "english_stop": { "type": "stop", "stopwords": "_english" }, "light_english_stemmer": { "type": "stemmer", "language": "light_english" <1> }, "english_possessive_stemmer": { "type": "stemmer", "language": "possessive_english" } }, "analyzer": { "english": { "tokenizer": "standard", "filter": [ "english_possessive_stemmer", "lowercase", "english_stop", "light_english_stemmer", <1> "asciifolding" <2> ] } } } }
```



- <1> Replaced the `english` stemmer with the less aggressive `light_english` stemmer
- <2> Added the `asciifolding` token filter



[[dictionary-stemmers]] === Dictionary Stemmers

Dictionary stemmers work quite differently from <>.((("stemming words", "dictionary stemmers")))((("dictionary stemmers"))) Instead of applying a standard set of rules to each word, they simply look up the word in the dictionary. Theoretically, they could produce much better results than an algorithmic stemmer. A dictionary stemmer should be able to do the following:

- Return the correct root word for irregular forms such as `feet` and `mice`
- Recognize the distinction between words that are similar but have different word senses —for example, `organ` and `organization`

In practice, a good algorithmic stemmer usually outperforms a dictionary stemmer. There are a couple of reasons this should be so:

Dictionary quality::

+

A dictionary stemmer is only as good as its dictionary. ((("dictionary stemmers", "dictionary quality and"))) The Oxford English Dictionary website estimates that the English language contains approximately 750,000 words (when inflections are included). Most English dictionaries available for computers contain about 10% of those.

The meaning of words changes with time. While stemming `mobility` to `mobil` may have made sense previously, it now conflates the idea of mobility with a mobile phone. Dictionaries need to be kept current, which is a time-consuming task. Often, by the time a dictionary has been made available, some of its entries are already out-of-date.

If a dictionary stemmer encounters a word not in its dictionary, it doesn't know how to deal with it. An algorithmic stemmer, on the other hand, will

apply the same rules as before, correctly or incorrectly.

Size and performance::

+



A dictionary stemmer needs to load all words,(("dictionary stemmers", "size and performance")) all prefixes, and all suffixes into memory. This can use a significant amount of RAM. Finding the right stem for a word is often considerably more complex than the equivalent process with an algorithmic stemmer.

Depending on the quality of the dictionary, the process of removing prefixes and suffixes may be more or less efficient. Less-efficient forms can slow the stemming process significantly.

Algorithmic stemmers, on the other hand, are usually simple, small, and fast.

TIP: If a good algorithmic stemmer exists for your language, it is usually a better choice than a dictionary-based stemmer. Languages with poor (or nonexistent) algorithmic stemmers can use the Hunspell dictionary stemmer, which we discuss in the next section.



[[hunspell]] === Hunspell Stemmer

Elasticsearch provides (((“dictionary stemmers”, “Hunspell stemmer”)))(((“stemming words”, “dictionary stemmers”, “Hunspell stemmer”)))dictionary-based stemming via the <http://bit.ly/1KNFdXI>[`hunspell` token filter]. Hunspell

<http://hunspell.sourceforge.net/>[hunspell.sourceforge.net] is the spell checker used by Open Office, LibreOffice, Chrome, Firefox, Thunderbird, and many other open and closed source projects.

Hunspell dictionaries((“Hunspell stemmer”, “obtaining a Hunspell dictionary”))) can be obtained from the following:

- <http://extensions.openoffice.org/>[extensions.openoffice.org]: Download and unzip the .oxt extension file.
- <http://mzl.la/157UORf>[[addons.mozilla.org](http://mzl.la/157UORf)]: Download and unzip the .xpi addon file.
- <http://bit.ly/1ygnODR>[OpenOffice archive]: Download and unzip the .zip file.

A Hunspell dictionary consists of two files with the same base name--such as en_us—but with one of two extensions:

.dic ::

Contains all the root words, in alphabetical order, plus a code representing all possible suffixes and prefixes (which collectively are known as _affixes_)

.aff ::

Contains the actual prefix or suffix transformation for each code listed in the `dic` file

==== Installing a Dictionary

The Hunspell token ((“Hunspell stemmer”, “installing a dictionary”)))filter looks for dictionaries within a dedicated Hunspell directory, which defaults to ./config/hunspell/. The .dic and .aff files should be placed in a subdirectory whose name represents the language or locale of the dictionaries. For instance, we could create a Hunspell stemmer for American English with the following layout:

[source, text]

```
config/ L hunspell/ <1> L en_US/ <2> | en_US.dic | en_US.aff
```



```
└ settings.yml <3>
```

<1> The location of the Hunspell directory can be changed by setting

```
indices.analysis.hunspell.dictionary.location
```

 in the `config/elasticsearch.yml` file.

<2> `en_US` will be the name of the locale or `language` that we pass to the `hunspell` token filter.

<3> Per-language settings file, described in the following section.

==== Per-Language Settings

The `settings.yml` file contains settings((("Hunspell stemmer", "per-language settings"))) that apply to all of the dictionaries within the language directory, such as these:

[source,yaml]

```
ignore_case: true strict_affix_parsing: true
```

The meaning of these settings is as follows:

```
ignore_case ::
```

+

Hunspell dictionaries are case sensitive by default: the surname `Booker` is a different word from the noun `booker`, and so should be stemmed differently. It may seem like a good idea to use the `hunspell` stemmer in case-sensitive mode,((("Hunspell stemmer", "using in case insensitive mode"))) but that can complicate things:

- A word at the beginning of a sentence will be capitalized, and thus appear to be a proper noun.
- The input text may be all uppercase, in which case almost no words will be found.
- The user may search for names in all lowercase, in which case no capitalized words will be found.

As a general rule, it is a good idea to set `ignore_case` to `true`.



```
strict_affix_parsing ::
```

The quality of dictionaries varies greatly.((("Hunspell stemmer", "strict_affix_parsing")))
Some dictionaries that are available online have malformed rules in the `.aff` file. By default, Lucene will throw an exception if it can't parse an affix rule. If you need to deal with a broken affix file, you can set `strict_affix_parsing` to `false` to tell Lucene to ignore the broken rules.((("strict_affix_parsing")))

.Custom Dictionaries

If multiple dictionaries (`.dic` files) are placed in the same directory, ((("Hunspell stemmer", "custom dictionaries"))), they will be merged together at load time. This allows you to tailor the downloaded dictionaries with your own custom word lists:

[source, text]

```
config/ └ hunspell/ └ en_US/ <1> ┌ en_US.dic ┌ en_US.aff <2> ┌ custom.dic
```

```
└ settings.yml
```

<1> The `custom` and `en_us` dictionaries will be merged.

<2> Multiple `.aff` files are not allowed, as they could use conflicting rules.

The format of the `.dic` and `.aff` files is discussed in <>.

==== Creating a Hunspell Token Filter

Once your dictionaries are installed on all nodes, you can define a `hunspell` token filter((("Hunspell stemmer", "creating a hunspell token filter")) that uses them:

[source, json]

```
PUT /my_index { "settings": { "analysis": { "filter": { "en_US": { "type": "hunspell", "language": "en_US" <1> } }, "analyzer": { "en_US": { "tokenizer": "standard", "filter": [ "lowercase", "en_US" ] } } } }
```



<1> The `language` has the same name as the directory where the dictionary lives.

You can test the new analyzer with the `analyze` API, and compare its output to that of the `english` analyzer:

[source,json]

```
GET /my_index/_analyze?analyzer=en_US <1> reorganizes
```

```
GET /_analyze?analyzer=english <2>
```

reorganizes

<1> Returns `organize`

<2> Returns `reorgan`

An interesting property of the `hunspell` stemmer, as can be seen in the preceding example, is that it can remove prefixes as well as suffixes. Most algorithmic stemmers remove suffixes only.

[TIP]

Hunspell dictionaries can consume a few megabytes of RAM. Fortunately, Elasticsearch creates only a single instance of a dictionary per node. All shards that use the same Hunspell analyzer share the same instance.

=====

[[hunspell-dictionary-format]] ===== Hunspell Dictionary Format

While it is not necessary to understand the(("Hunspell stemmer", "Hunspell dictionary format")) format of a Hunspell dictionary in order to use the `hunspell` tokenizer, understanding the format will help you write your own custom dictionaries. It is quite simple.

For instance, in the US English dictionary, the `en_us.dic` file contains an entry for the word `analyze`, which looks like this:



[source, text]

analyze/ADSG

The `en_US.aff` file contains the prefix or suffix rules for the `A`, `G`, `D`, and `S` flags. Each flag consists of a number of rules, only one of which should match. Each rule has the following format:

[source, text]

[type] [flag] [letters to remove] [letters to add] [condition]

For instance, the following is suffix (`SFX`) rule `D`. It says that, when a word ends in a consonant (anything but `a`, `e`, `i`, `o`, or `u`) followed by a `y`, it can have the `y` removed and `ied` added (for example, `ready` -> `readied`).

[source, text]

SFX D y ied aeiouy

The rules for the `A`, `G`, `D`, and `S` flags mentioned previously are as follows:

[source, text]

SFX D Y 4 SFX D 0 d e <1> SFX D y ied **aeiouy** SFX D 0 ed **ey** SFX D 0 ed [aeiou]y

SFX S Y 4 SFX S y ies **aeiouy** SFX S 0 s [aeiou]y SFX S 0 es [sxzh] SFX S 0 s **sxzh** <2>

SFX G Y 2 SFX G e ing e <3> SFX G 0 ing **e**

PFX A Y 1

PFX A 0 re . <4>



<1> `analyze` ends in an `e`, so it can become `analyzed` by adding a `d`.

<2> `analyze` does not end in `s`, `x`, `z`, `h`, or `y`, so it can become `analyzes` by adding an `s`.

<3> `analyze` ends in an `e`, so it can become `analyzing` by removing the `e` and adding `ing`.

<4> The prefix `re` can be added to form `reanalyze`. This rule can be combined with the suffix rules to form `reanalyzes`, `reanalyzed`, `reanalyzing`.

More information about the Hunspell syntax can be found on the [Hunspell documentation site](http://bit.ly/1ynGhv6).



[[choosing-a-stemmer]] === Choosing a Stemmer

The documentation for the <http://bit.ly/1AUfpDN>[`stemmer`] token filter lists multiple stemmers for some languages.((("stemming words", "choosing a stemmer")))((("English", "stemmers for"))) For English we have the following:

english ::

The <http://bit.ly/17LseXy>[`porter_stem`] token filter.

light_english ::

The <http://bit.ly/1I0bUjZ>[`kstem`] token filter.

minimal_english ::

The `EnglishMinimalStemmer` in Lucene, which removes plurals

lovins ::

The <http://bit.ly/1Cr4tNI>[Snowball] based
<http://bit.ly/1ICyTjR>[Lovins]
stemmer, the first stemmer ever produced.

porter ::

The <http://bit.ly/1Cr4tNI>[Snowball] based
<http://bit.ly/1sCwiwj>[Porter] stemmer

porter2 ::

The <http://bit.ly/1Cr4tNI>[Snowball] based
<http://bit.ly/1zip3lK>[Porter2] stemmer

possessive_english ::

The `EnglishPossessiveFilter` in Lucene which removes ``s``

Add to that list the Hunspell stemmer with the various English dictionaries that are available.



One thing is for sure: whenever more than one solution exists for a problem, it means that none of the solutions solves the problem adequately. This certainly applies to stemming -- each stemmer uses a different approach that overstems and understems words to a different degree.

The `stemmer` documentation page ((("languages", "stemmers for"))) highlights the recommended stemmer for each language in bold, usually because it offers a reasonable compromise between performance and quality. That said, the recommended stemmer may not be appropriate for all use cases. There is no single right answer to the question of which is the best stemmer -- it depends very much on your requirements. There are three factors to take into account when making a choice: performance, quality, and degree.

[[stemmer-performance]] === Stemmer Performance

Algorithmic stemmers are typically four or ((("stemming words", "choosing a stemmer", "stemmer performance"))) ((("Hunspell stemmer", "performance"))) five times faster than Hunspell stemmers. `Handcrafted` algorithmic stemmers are usually, but not always, faster than their Snowball equivalents. For instance, the `porter_stem` token filter` is significantly faster than the Snowball implementation of the Porter stemmer.

Hunspell stemmers have to load all words, prefixes, and suffixes into memory, which can consume a few megabytes of RAM. Algorithmic stemmers, on the other hand, consist of a small amount of code and consume very little memory.

[[stemmer-quality]] === Stemmer Quality

All languages, except Esperanto, are irregular.((("stemming words", "choosing a stemmer", "stemmer quality"))) While more-formal words tend to follow a regular pattern, the most commonly used words often have irregular rules. Some stemming algorithms have been developed over years of research and produce reasonably high-quality results. Others have been assembled more quickly with less research and deal only with the most common cases.

While Hunspell offers the promise of dealing precisely with irregular words, it often falls short in practice. A dictionary stemmer is only as good as its dictionary. If Hunspell comes across a word that isn't in its dictionary, it can do nothing with it. Hunspell requires an extensive, high-quality, up-to-date dictionary in order to produce good results; dictionaries of this caliber are few and far between. An algorithmic stemmer, on the other hand, will happily deal with new words that didn't exist when the designer created the algorithm.

If a good algorithmic stemmer is available for your language, it makes sense to use it rather than Hunspell. It will be faster, will consume less memory, and will generally be as good or better than the Hunspell equivalent.



If accuracy and customizability is important to you, and you need (and have the resources) to maintain a custom dictionary, then Hunspell gives you greater flexibility than the algorithmic stemmers. (See <> for customization techniques that can be used with any stemmer.)

[[stemmer-degree]] ===== Stemmer Degree

Different stemmers overstem and understem(("stemming words", "choosing a stemmer", "stemmer degree")) to a different degree. The `light_` stemmers stem less aggressively than the standard stemmers, and the `minimal_` stemmers less aggressively still. Hunspell stems aggressively.

Whether you want aggressive or light stemming depends on your use case. If your search results are being consumed by a clustering algorithm, you may prefer to match more widely (and, thus, stem more aggressively). If your search results are intended for human consumption, lighter stemming usually produces better results. Stemming nouns and adjectives is more important for search than stemming verbs, but this also depends on the language.

The other factor to take into account is the size of your document collection. With a small collection such as a catalog of 10,000 products, you probably want to stem more aggressively to ensure that you match at least some documents. If your collection is large, you likely will get good matches with lighter stemming.

===== Making a Choice

Start out with a recommended stemmer. If it works well enough, there is no need to change it. If it doesn't, you will need to spend some time investigating and comparing the stemmers available for language in order to find the one that best suits your purposes.



[[controlling-stemming]] === Controlling Stemming

Out-of-the-box stemming solutions are never perfect.((("stemming words", "controlling stemming"))) Algorithmic stemmers, especially, will blithely apply their rules to any words they encounter, perhaps conflating words that you would prefer to keep separate. Maybe, for your use case, it is important to keep `skies` and `skiing` as distinct words rather than stemming them both down to `ski` (as would happen with the `english` analyzer).

The <http://bit.ly/1IOeXZD>[`keyword_marker`] and <http://bit.ly/1ymcioJ>[`stemmer_override`] token filters((("stemmer_override token filter")))((("keyword_marker token filter"))) allow us to customize the stemming process.

[[preventing-stemming]] === Preventing Stemming

The `<1>` parameter for language analyzers (see `<1>`) allowed ((("stemming words", "controlling stemming", "preventing stemming"))) us to specify a list of words that should not be stemmed. Internally, these language analyzers use the <http://bit.ly/1IOeXZD>[`keyword_marker` token filter] to mark the listed words as *keywords*, which prevents subsequent stemming token filters from touching those words. (((("keyword_marker token filter", "preventing stemming of certain words"))))

For instance, we can create a simple custom analyzer that uses the <http://bit.ly/17LseXy>[`porter_stem`] token filter, but prevents the word `skies` from(((("porter_stem token filter")))) being stemmed:

[source,json]

```
PUT /my_index { "settings": { "analysis": { "filter": { "no_stem": { "type": "keyword_marker", "keywords": [ "skies" ] <1> } }, "analyzer": { "my_english": { "tokenizer": "standard", "filter": [ "lowercase", "no_stem", "porter_stem" ] } } } }
```

```
}
```

<1> They `keywords` parameter could accept multiple words.

Testing it with the `analyze` API shows that just the word `skies` has been excluded from stemming:

[source,json]

```
GET /my_index/_analyze?analyzer=my_english
```



sky skies skiing skis <1>

<1> Returns: sky , skies , ski , ski

[[keyword-path]]

[TIP]

While the language analyzers allow (((("language analyzers", "stem_exclusion parameter"))))us only to specify an array of words in the `stem_exclusion` parameter, the `keyword_marker` token filter also accepts a `keywords_path` parameter that allows us to store all of our keywords in a file. (((("keyword_marker token filter", "keywords_path parameter"))))The file should contain one word per line, and must be present on every node in the cluster. See <> for tips on how to update this file.

=====

[[customizing-stemming]] ===== Customizing Stemming

In the preceding example, we prevented `skies` from being stemmed, but perhaps we would prefer it to be stemmed to `sky` instead.(((("stemming words", "controlling stemming", "customizing stemming")))) The <http://bit.ly/1ymcioJ>[`stemmer_override`] token filter allows us (((("stemmer_override token filter"))))to specify our own custom stemming rules. At the same time, we can handle some irregular forms like stemming `mice` to `mouse` and `feet` to `foot` :

[source,json]

```
PUT /my_index { "settings": { "analysis": { "filter": { "custom_stem": { "type": "stemmer_override", "rules": [ <1> "skies=>sky", "mice=>mouse", "feet=>foot" ] } }, "analyzer": { "my_english": { "tokenizer": "standard", "filter": [ "lowercase", "custom_stem", <2> "porter_stem" ] } } } }
```

```
GET /my_index/_analyze?analyzer=my_english
```

The mice came down from the skies and ran over my feet <3>

<1> Rules take the form `original=>stem` .



<2> The `stemmer_override` filter must be placed before the stemmer.

<3> Returns `the , mouse , came , down , from , the , sky , and , ran , over , my , foot .`

TIP: Just as for the `keyword_marker` token filter, rules can be stored in a file whose location should be specified with the `rules_path` parameter.



[[stemming-in-situ]] === Stemming in situ

For the sake of completeness, we will ((("stemming words", "stemming in situ"))) finish this chapter by explaining how to index stemmed words into the same field as unstemmed words. As an example, analyzing the sentence *The quick foxes jumped* would produce the following terms:

[source, text]

Pos 1: (the) Pos 2: (quick) Pos 3: (foxes,fox) <1>

Pos 4: (jumped,jump) <1>

<1> The stemmed and unstemmed forms occupy the same position.

WARNING: Read <> before using this approach.

To achieve stemming *in situ*, we will use the <http://bit.ly/1ynIBCe> [`keyword_repeat`] token filter, ((("keyword_repeat token filter"))) which, like the `keyword_marker` token filter (see <>), marks each term as a keyword to prevent the subsequent stemmer from touching it. However, it also repeats the term in the same position, and this repeated term *is* stemmed.

Using the `keyword_repeat` token filter alone would result in the following:

[source, text]

Pos 1: (the,the) <1> Pos 2: (quick,quick) <1> Pos 3: (foxes,fox)

Pos 4: (jumped,jump)

<1> The stemmed and unstemmed forms are the same, and so are repeated needlessly.

To prevent the useless repetition of terms that are the same in their stemmed and unstemmed forms, we add the <http://bit.ly/1B6xHUY> [`unique`] token filter ((("unique token filter"))) into the mix:

[source, json]



```
PUT /my_index { "settings": { "analysis": { "filter": { "unique_stem": { "type": "unique", "only_on_same_position": true <1> } }, "analyzer": { "in_situ": { "tokenizer": "standard", "filter": [ "lowercase", "keyword_repeat", <2> "porter_stem", "unique_stem" <3> ] } } } }
```

}

<1> The `unique` token filter is set to remove duplicate tokens only when they occur in the same position.

<2> The `keyword_repeat` token filter must appear before the stemmer.

<3> The `unique_stem` filter removes duplicate terms after the stemmer has done its work.

[[stemming-in-situ-good-idea]] ===== Is Stemming in situ a Good Idea

People like the (((("stemming words", "stemming in situ", "good idea, or not"))))idea of stemming *in situ*: ``Why use an unstemmed field *and* a stemmed field if I can just use one combined field?'' But is it a good idea? The answer is almost always no. There are two problems.

The first is the inability to separate exact matches from inexact matches. In this chapter, we have seen that words with different meanings are often conflated to the same stem word: `organs` and `organization` both stem to `organ`.

In <>, we demonstrated how to combine a query on a stemmed field (to increase recall) with a query on an unstemmed field (to improve relevance).(((("language analyzers", "combining query on stemmed and unstemmed field")))) When the stemmed and unstemmed fields are separate, the contribution of each field can be tuned by boosting one field over another (see <>). If, instead, the stemmed and unstemmed forms appear in the same field, there is no way to tune your search results.

The second issue has to do with how the (((("relevance scores", "stemming in situ and"))))relevance score is calculated. In <>, we explained that part of the calculation depends on the *inverse document frequency* -- how often a word appears in all the documents in our index.(((("inverse document frequency", "stemming in situ and")))) Using *in situ* stemming for a document that contains the text `jump jumped jumps` would result in these terms:

[source, text]

Pos 1: (`jump`) Pos 2: (`jumped,jump`)



Pos 3: (jumps,jump)

While `jumped` and `jumps` appear once each and so would have the correct IDF, `jump` appears three times, greatly reducing its value as a search term in comparison with the unstemmed forms.

For these reasons, we recommend against using stemming in situ.



[[stopwords]] == Stopwords: Performance Versus Precision

Back in the early days of information retrieval,(("stopwords", "performance versus precision"))) disk space and memory were limited to a tiny fraction of what we are accustomed to today. It was essential to make your index as small as possible. Every kilobyte saved meant a significant improvement in performance. Stemming (see <>) was important, not just for making searches broader and increasing retrieval in the same way that we use it today, but also as a tool for compressing index size.

Another way to reduce index size is simply to *index fewer words*. For search purposes, some words are more important than others. A significant reduction in index size can be achieved by indexing only the more important terms.

So which terms can be left out?(("term frequency", "high and low")) We can divide terms roughly into two groups:

Low-frequency terms::

Words that appear in relatively few documents in the collection. Because of their rarity,(("weight", "low frequency terms"))) they have a high value, or *weight*.

High-frequency terms::

Common words that appear in many documents in the index, such as the , and , and is . These words have a low weight and contribute little to the relevance score.

[TIP]

Of course, frequency is really a scale rather than just two points labeled *low* and *high*. We just draw a line at some arbitrary point and say that any terms below that line are low frequency and above the line are high frequency.

Which terms are low or high frequency depend on the documents themselves. The word and may be a low-frequency term if all the documents are in Chinese. In a collection of documents about databases, the word database may be a high-frequency term with little value as a search term for that particular collection.

That said, for any language there are words that occur very commonly and that seldom add value to a search.(("English", "stopwords")) The default English stopwords used in Elasticsearch are as follows:



a, an, and, are, as, at, be, but, by, for, if, in, into, is, it, no, not, of, on, or, such, that, the, their, then, there, these, they, this, to, was, will, with

These *stopwords* can usually be filtered out before indexing with little negative impact on retrieval. But is it a good idea to do so?

[[pros-cons-stopwords]] [float="true"] === Pros and Cons of Stopwords

We have more disk space, more RAM, and ((("stopwords", "pros and cons of"))) better compression algorithms than existed back in the day. Excluding the preceding 33 common words from the index will save only about 4MB per million documents. Using stopwords for the sake of reducing index size is no longer a valid reason. (However, there is one caveat to this statement, which we discuss in <>.)

On top of that, by removing words from the index, we are reducing our ability to perform certain types of searches. Filtering out the words listed previously prevents us from doing the following:

- Distinguishing *happy* from *not happy*.
- Searching for the band The The.
- Finding Shakespeare's quotation ``To be, or not to be''
- Using the country code for Norway: no

The primary advantage of removing stopwords is performance. Imagine that we search an index with one million documents for the word `fox`. Perhaps `fox` appears in only 20 of them, which means that Elasticsearch has to calculate the relevance `_score` for 20 documents in order to return the top 10. Now, we change that to a search for `the OR fox`. The word `the` probably occurs in almost all the documents, which means that Elasticsearch has to calculate the `_score` for all one million documents. This second query simply cannot perform as well as the first.

Fortunately, there are techniques that we can use to keep common words searchable, while still maintaining good performance. First, we'll start with how to use stopwords.



[[using-stopwords]] === Using Stopwords

The removal of stopwords is ((("stopwords", "removal of"))) handled by the <http://bit.ly/1INX4tN> [`stop` token filter] which can be used when ((("stop token filter"))) creating a `custom` analyzer (see <>). However, some out-of-the-box analyzers((("analyzers", "stop filter pre-integrated")))(((("pattern analyzer", "stopwords and"))))((("standard analyzer", "stop filter")))(((("language analyzers", "stop filter pre-integrated")))) come with the `stop` filter pre-integrated:

<http://bit.ly/1xtdoJV> [Language analyzers]:

Each language analyzer defaults to using the appropriate stopwords list for that language. For instance, the `english` analyzer uses the `_english_\` stopwords list.

<http://bit.ly/14EpXv3> [`standard` analyzer]:

Defaults to the empty stopwords list: `_none_\`, essentially disabling stopwords.

<http://bit.ly/1u9OVct> [`pattern` analyzer]:

Defaults to `_none_\`, like the `standard` analyzer.

==== Stopwords and the Standard Analyzer

To use custom stopwords in conjunction with ((("standard analyzer", "stopwords and")))(((("stopwords", "using with standard analyzer")))) the `standard` analyzer, all we need to do is to create a configured version of the analyzer and pass in the list of `stopwords` that we require:

[source,json]

```
PUT /my_index { "settings": { "analysis": { "analyzer": { "my_analyzer": { <1> "type": "standard", <2> "stopwords": [ "and", "the" ] <3> } } } }
```

```
}
```

<1> This is a custom analyzer called `my_analyzer`.

<2> This analyzer is the `standard` analyzer with some custom configuration.



<3> The stopwords to filter out are `and` and `the`.

TIP: This same technique can be used to configure custom stopword lists for any of the language analyzers.

[[maintaining-positions]] ===== Maintaining Positions

The output from the `analyze` API(((`"stopwords"`, "maintaining position of terms and")))) is quite interesting:

[source,json]

```
GET /my_index/_analyze?analyzer=my_analyzer
```

The quick and the dead

[source,json]

```
{ "tokens": [ { "token": "quick", "start_offset": 4, "end_offset": 9, "type": "", "position": 2 <1> }, { "token": "dead", "start_offset": 18, "end_offset": 22, "type": "", "position": 5 <1> } ] }
```

```
}
```

<1> Note the `position` of each token.

The stopwords have been filtered out, as expected, but the interesting part is that the `position` of the(((`"phrase matching"`, `"stopwords and"`, `"positions data"`))) two remaining terms is unchanged: `quick` is the second word in the original sentence, and `dead` is the fifth. This is important for phrase queries--if the positions of each term had been adjusted, a phrase query for `quick dead` would have matched the preceding example incorrectly.

[[specifying-stopwords]] ===== Specifying Stopwords

Stopwords can be passed inline, as we did in (((`"stopwords"`, `"specifying"`))) the previous example, by specifying an array:

[source,json]

```
"stopwords": [ "and", "the" ]
```



The default stopword list for a particular language can be specified using the `_lang_` notation:

[source,json]

"stopwords": "english"

TIP: The predefined language-specific stopword((("languages", "predefined stopword lists for"))) lists available in Elasticsearch can be found in the [\[`stop` token filter\]](http://bit.ly/157YLFy) documentation.

Stopwords can be disabled by ((("stopwords", "disabling")))specifying the special list: `_none_`. For instance, to use the `english` analyzer((("english analyzer", "using without stopwords"))) without stopwords, you can do the following:

[source,json]

```
PUT /myindex { "settings": { "analysis": { "analyzer": { "my_english": { "type": "english", <1>  
"stopwords": "_none" <2> } } } }
```

```
}
```

<1> The `my_english` analyzer is based on the `english` analyzer.

<2> But stopwords are disabled.

Finally, stopwords can also be listed in a file with one word per line. The file must be present on all nodes in the cluster, and the path can be specified((("stopwords_path parameter"))) with the `stopwords_path` parameter:

[source,json]

```
PUT /my_index { "settings": { "analysis": { "analyzer": { "my_english": { "type": "english",  
"stopwords_path": "stopwords/english.txt" <1> } } } }
```

```
}
```



<1> The path to the stopwords file, relative to the Elasticsearch `config` directory

[[stop-token-filter]] ===== Using the stop Token Filter

The <http://bit.ly/1AUzDNI> [`stop` token filter] can be combined with a tokenizer(((`"stopwords", "using stop token filter"`)))((`"stop token filter", "using in custom analyzer"`))) and other token filters when you need to create a `custom` analyzer. For instance, let's say that we wanted to ((`"Spanish", "custom analyzer for"`))((`"light_spanish stemmer"`)))create a Spanish analyzer with the following:

- A custom stopwords list
- The `light_spanish` stemmer
- The <> to remove diacritics

We could set that up as follows:

[source,json]

```
PUT /my_index { "settings": { "analysis": { "filter": { "spanish_stop": { "type": "stop", "stopwords": [ "si", "esta", "el", "la" ] <1> }, "light_spanish": { <2> "type": "stemmer", "language": "light_spanish" } }, "analyzer": { "my_spanish": { "tokenizer": "spanish", "filter": [ <3> "lowercase", "asciifolding", "spanish_stop", "light_spanish" ] } } } }
```

}

<1> The `stop` token filter takes the same `stopwords` and `stopwords_path` parameters as the `standard` analyzer.

<2> See <>.

<3> The order of token filters is important, as explained next.

We have placed the `spanish_stop` filter after the `asciifolding` filter.((`"asciifolding token filter", "in custom Spanish analyzer"`))) This means that `esta` , `ésta` , and `++está++` will first have their diacritics removed to become just `esta` , which will then be removed as a stopword. If, instead, we wanted to remove `esta` and `ésta` , but not `++está++`, we would have to put the `spanish_stop` filter *before* the `asciifolding` filter, and specify both words in the stopwords list.

[[updating-stopwords]] ===== Updating Stopwords



A few techniques can be used to update the list of stopwords used by an analyzer.

((("analyzers", "stopwords list, updating")))((("stopwords", "updating list used by analyzers"))))

Analyzers are instantiated at index creation time, when a node is restarted, or when a closed index is reopened.

If you specify stopwords inline with the `stopwords` parameter, your only option is to close the index and update the analyzer configuration with the [http://bit.ly/1zijFPx\[update index settings API\]](http://bit.ly/1zijFPx[update index settings API]), then reopen the index.

Updating stopwords is easier if you specify them in a file with the `stopwords_path` parameter.((("stopwords_path parameter"))) You can just update the file (on every node in the cluster) and then force the analyzers to be re-created by either of these actions:

- Closing and reopening the index (see [http://bit.ly/1B6s0WY\[open/close index\]](http://bit.ly/1B6s0WY[open/close index])), or
- Restarting each node in the cluster, one by one

Of course, updating the stopwords list will not change any documents that have already been indexed. It will apply only to searches and to new or updated documents. To apply the changes to existing documents, you will need to reindex your data. See <>.



[[stopwords-performance]] === Stopwords and Performance

The biggest disadvantage of keeping stopwords is that of performance. When Elasticsearch performs a (((("stopwords", "performance and"))))full-text search, it has to calculate the relevance `_score` on all matching documents in order to return the top 10 matches.

While most words typically occur in much fewer than 0.1% of all documents, a few words such as `the` may occur in almost all of them. Imagine you have an index of one million documents. A query for `quick brown fox` may match fewer than 1,000 documents. But a query for `the quick brown fox` has to score and sort almost all of the one million documents in your index, just in order to return the top 10!

The problem is that `the quick brown fox` is really a query for `the OR quick OR brown OR fox` —any document that contains nothing more than the almost meaningless term `the` is included in the result set. What we need is a way of reducing the number of documents that need to be scored.

[[stopwords-and]] === and Operator

The easiest way to reduce the number of documents is simply to use the `<>` with the `match` query, in order to make all (((("stopwords", "performance and", "using and operator")))((("and operator", "using with match query"))))words required.

A `match` query like this:

[source,json]

```
{ "match": { "text": { "query": "the quick brown fox", "operator": "and" } } }
```

```
}
```

is rewritten as a `bool` query like this:

[source,json]

```
{ "bool": { "must": [ { "term": { "text": "the" }}, { "term": { "text": "quick" }}, { "term": { "text": "brown" }}, { "term": { "text": "fox" } } ] }
```

```
}
```

The `bool` query is intelligent enough to execute each `term` query in the optimal order--it starts with the least frequent term. Because all terms are required, only documents that contain the least frequent term can possibly match. Using the `and` operator greatly speeds up multiterm queries.

===== `minimum_should_match`

In <>, we discussed using the `minimum_should_match` operator to trim the long tail of less-relevant results.((("stopwords", "performance and", "using minimum_should_match operator")))((("minimum_should_match parameter"))) It is useful for this purpose alone but, as a nice side effect, it offers a similar performance benefit to the `and` operator:

[source,json]

```
{ "match": { "text": { "query": "the quick brown fox", "minimum_should_match": "75%" } }}
```

```
}
```

In this example, at least three out of the four terms must match. This means that the only docs that need to be considered are those that contain either the least or second least frequent terms.

This offers a huge performance gain over a simple query with the default `or` operator! But we can do better yet...



[[common-terms]] === Divide and Conquer

The terms in a query string can be divided into more-important (low-frequency) and less-important (high-frequency) terms.((("stopwords", "low and high frequency terms")))

Documents that match only the less important terms are probably of very little interest.

Really, we want documents that match as many of the more important terms as possible.

The `match` query accepts (((("cutoff_frequency parameter"))))((("match query", "cutoff_frequency parameter"))))a `cutoff_frequency` parameter, which allows it to divide the terms in the query string into a low-frequency and high-frequency group.(((("term frequency", "cutoff_frequency parameter in match query")))) The low-frequency group (more-important terms) form the bulk of the query, while the high-frequency group (less-important terms) is used only for scoring, not for matching. By treating these two groups differently, we can gain a real boost of speed on previously slow queries.

.Domain-Specific Stopwords

One of the benefits of `cutoff_frequency` is that you get *domain-specific* stopwords for free.(((("domain specific stopwords"))))((("stopwords", "domain specific"))) For instance, a website about movies may use the words *movie*, *color*, *black*, and *white* so often that they could be considered almost meaningless. With the `stop` token filter, these domain-specific terms would have to be added to the stopwords list manually. However, because the `cutoff_frequency` looks at the actual frequency of terms in the index, these words would be classified as *high frequency* automatically.

Take this query as an example:

[source,json]

```
{ "match": { "text": { "query": "Quick and the dead", "cutoff_frequency": 0.01 <1> } }
```

```
}
```

<1> Any term that occurs in more than 1% of documents is considered to be high frequency.

The `cutoff_frequency` can be specified as a fraction (`0.01`) or as an absolute number (`5`).



This query uses the `cutoff_frequency` to first divide the query terms into a low-frequency group (`quick`, `dead`) and a high-frequency group (`and`, `the`). Then, the query is rewritten to produce the following `bool` query:

[source.json]

```
{ "bool": { "must": { <1> "bool": { "should": [ { "term": { "text": "quick" }}, { "term": { "text": "dead" } } ] } }, "should": { <2> "bool": { "should": [ { "term": { "text": "and" }}, { "term": { "text": "the" } } ] } } }
```

```
}
```

<1> At least one low-frequency/high-importance term *must* match.

<2> High-frequency/low-importance terms are entirely optional.

The `must` clause means that at least one of the low-frequency terms—`quick` or `dead`—*must* be present for a document to be considered a match. All other documents are excluded. The `should` clause then looks for the high-frequency terms `and` and `the`, but only in the documents collected by the `must` clause. The sole job of the `should` clause is to score a document like `Quick_and the_ dead'` higher than `The quick but dead'`. This approach greatly reduces the number of documents that need to be examined and scored.

[TIP]

Setting the operator parameter to `and` would make *all* low-frequency terms required, and score documents that contain *all* high-frequency terms higher. However, matching documents would not be required to contain all high-frequency terms. If you would prefer all low- and high-frequency terms to be required, you should use a `bool` query instead. As we saw in <>, this is already an efficient query.

=====

==== Controlling Precision

The `minimum_should_match` parameter can be combined with `cutoff_frequency` but it applies to only the low-frequency terms.((("stopwords", "low and high frequency terms", "controlling precision")))((("minimum_should_match parameter", "controlling precision")))) This query:



[source,json]

```
{ "match": { "text": { "query": "Quick and the dead", "cutoff_frequency": 0.01, "minimum_should_match": "75%" }}
```

```
}
```

would be rewritten as follows:

[source,json]

```
{ "bool": { "must": { "bool": { "should": [ { "term": { "text": "quick" }}, { "term": { "text": "dead" }} ], "minimum_should_match": 1 <1> }, "should": { <2> "bool": { "should": [ { "term": { "text": "and" }}, { "term": { "text": "the" }} ] } } }}
```

```
}
```

<1> Because there are only two terms, the original 75% is rounded down to `1`, that is: *one out of two low-terms must match*.

<2> The high-frequency terms are still optional and used only for scoring.

==== Only High-Frequency Terms

An `or` query for high-frequency((("stopwords", "low and high frequency terms", "only high frequency terms"))) terms only—``To be, or not to be''—is the worst case for performance. It is pointless to score *all* the documents that contain only one of these terms in order to return just the top 10 matches. We are really interested only in documents in which the terms all occur together, so in the case where there are no low-frequency terms, the query is rewritten to make all high-frequency terms required:

[source,json]

```
{ "bool": { "must": [ { "term": { "text": "to" }}, { "term": { "text": "be" }}, { "term": { "text": "or" }}, { "term": { "text": "not" }}, { "term": { "text": "to" }}, { "term": { "text": "be" }} ] }
```

```
}
```



==== More Control with Common Terms

While the high/low frequency functionality in the `match` query is useful, sometimes you want more control(((`"stopwords"`, `"low and high frequency terms"`, `"more control over common terms"`))) over how the high- and low-frequency groups should be handled. The `match` query exposes a subset of the functionality available in the `common terms` query.(((`"common terms query"`)))

For instance, we could make all low-frequency terms required, and score only documents that have 75% of all high-frequency terms with a query like this:

[source.json]

```
{ "common": { "text": { "query": "Quick and the dead", "cutoff_frequency": 0.01,  
"low_freq_operator": "and", "minimum_should_match": { "high_freq": "75%" } } }
```

```
}
```

See the <http://bit.ly/1wdS2Qo>[`common` terms query] reference page for more options.



[[stopwords-phrases]] === Stopwords and Phrase Queries

About 5% of all queries are (((**stopwords**, "phrase queries and")))((("phrase matching", "**stopwords and**")))**phrase queries** (see <>), but they often account for the majority of slow queries. Phrase queries can perform poorly, especially if the phrase includes very common words; a phrase like ``To be, or not to be'' could be considered pathological. The reason for this has to do with the amount of data that is necessary to support proximity matching.

In <>, we said that removing stopwords saves only a small amount of space in the inverted index.(((**indices**, "typical, data contained in"))) That was only partially true. A typical index may contain, among other data, some or all of the following:

Terms dictionary::

A sorted list of all terms that appear in the documents in the index, and a count of the number of documents that contain each term.

Postings list::

A list of which documents contain each term.

Term frequency::

How often each term appears in each document.

Positions::

The position of each term within each document, for phrase and proximity queries.

Offsets::

The start and end character offsets of each term in each document, for snippet highlighting. Disabled by default.

Norms::

A factor used to normalize fields of different lengths, to give shorter fields more weight.



Removing stopwords from the index may save a small amount of space in the *terms dictionary* and the *postings list*, but *positions* and *offsets* are another matter. Positions and offsets data can easily double, triple, or quadruple index size.

==== Positions Data

Positions are enabled on `analyzed` string fields by default,((("stopwords", "phrase queries and", "positions data")))((("phrase matching", "stopwords and", "positions data")))) so that phrase queries will work out of the box. The more often that a term appears, the more space is needed to store its position data. Very common words, by definition, appear very commonly, and their positions data can run to megabytes or gigabytes on large collections.

Running a phrase query on a high-frequency word like `the` might result in gigabytes of data being read from disk. That data will be stored in the kernel filesystem cache to speed up later access, which seems like a good thing, but it might cause other data to be evicted from the cache, which will slow subsequent queries.

This is clearly a problem that needs solving.

[[index-options]] === Index Options

The first question you should ((("stopwords", "phrase queries and", "index options")))((("phrase matching", "stopwords and", "index options"))))ask yourself is: *Do you need phrase or proximity queries?*

Often, the answer is no. For many use cases, such as logging, you need to know *whether* a term appears in a document -- information that is provided by the postings list--but not *where* it appears. Or perhaps you need to use phrase queries on one or two fields, but you can disable positions data on all of the other analyzed `string` fields.

The `index_options` parameter ((("index_options parameter"))))allows you to control what information is stored in the index for each field.((("fields", "index options")))) Valid values are as follows:

`docs` ::

Only store which documents contain which terms. This is the default for `not_analyzed` string fields.

`freqs` ::

Store `docs` information, plus how often each term appears in each document. Term frequencies are needed for complete <>relevance-intro,TF/IDF>> relevance calculations, but they are not required if you just need to know whether a document contains a particular term.



positions ::

```
Store `docs` and `freqs`, plus the position of each term in each document.  
This is the default for `analyzed` string fields, but can be disabled if  
phrase/proximity matching is not needed.
```

offsets ::

```
Store `docs`, `freqs`, `positions`, and the start and end character offsets  
of each term in the original string. This information is used by the  
http://bit.ly/1u9PJ16[`postings` highlighter]  
but is disabled by default.
```

You can set `index_options` on fields added at index creation time, or when adding new fields by using(((`put-mapping API`))) the `put-mapping` API. This setting can't be changed on existing fields:

[source,json]

```
PUT /my_index { "mappings": { "my_type": { "properties": { "title": { <1> "type": "string" },  
"content": { <2> "type": "string", "index_options": "freqs" } } } }
```

}

<1> The `title` field uses the default setting of `positions`, so it is suitable for phrase/proximity queries.

<2> The `content` field has positions disabled and so cannot be used for phrase/proximity queries.

==== Stopwords

Removing stopwords is one way of reducing the size of the positions data quite dramatically. (((`stopwords`, "phrase queries and", "removing stopwords")))) An index with stopwords removed can still be used for phrase queries because the original positions of the remaining terms are maintained, as we saw in <>. But of course, excluding terms from the index reduces searchability. We wouldn't be able to differentiate between the two phrases *Man in the moon* and *Man on the moon*.

Fortunately, there is a way to have our cake and eat it: the <>.



[[common-grams]] === common_grams Token Filter

The `common_grams` token filter is designed to make phrase queries with stopwords more efficient. (((`stopwords`, "phrase queries and", "`commongrams token filter`")))((`common_grams token filter`))((`phrase matching`, "stopwords and", "`common_grams token filter`"))*It is similar to the `shingles` token (((`shingles`, "shingles token filter")`)filter` (see <>), which creates _bigrams out of every pair of adjacent words. It is most easily explained by example.(((`bigrams`)))*

The `common_grams` token filter produces different output depending on whether `query_mode` is set to `false` (for indexing) or to `true` (for searching), so we have to create two separate analyzers:

[source,json]

```
PUT /myindex { "settings": { "analysis": { "filter": { "index_filter": { <1> "type": "common_grams", "common_words": "_english" <2> }, "searchfilter": { <1> "type": "common_grams", "common_words": "_english", <2> "query_mode": true } }, "analyzer": { "index_grams": { <3> "tokenizer": "standard", "filter": [ "lowercase", "index_filter" ] }, "search_grams": { <3> "tokenizer": "standard", "filter": [ "lowercase", "search_filter" ] } } } }
```

<1> First we create two token filters based on the `common_grams` token filter: `index_filter` for index time (with `query_mode` set to the default `false`), and `search_filter` for query time (with `query_mode` set to `true`).

<2> The `common_words` parameter accepts the same options as the `stopwords` parameter (see <>). The filter also accepts a `common_words_path` parameter, which allows you to maintain the common words list in a file.

<3> Then we use each filter to create an analyzer for index time and another for query time.

With our custom analyzers in place, we can create a field that will use the `index_grams` analyzer at index time:

[source,json]

```
PUT /my_index/_mapping/my_type { "properties": { "text": { "type": "string", "index_analyzer": "index_grams", <1> "search_analyzer": "standard" <1> } } }
```



<1> The `text` field uses the `index_grams` analyzer at index time, but defaults to using the standard analyzer at search time, for reasons we will explain next.

==== At Index Time

If we were to (((`"commongrams token filter", "at index time"`)))analyze the phrase `_The quick and brown fox` with shingles, it would produce these terms:

[source, text]

Pos 1: `the_quick` Pos 2: `quick_and` Pos 3: `and_brown`

Pos 4: `brown_fox`

Our new `index_grams` analyzer produces the following terms instead:

[source, text]

Pos 1: `the, the_quick` Pos 2: `quick, quick_and` Pos 3: `and, and_brown` Pos 4: `brown`

Pos 5: `fox`

All terms are output as unigrams—`the` , `quick` , and so forth--but if a word is a common word or is followed by a common word, then it also outputs a bigram in the same position as the unigram—`the_quick` , `quick_and` , `and_brown` .

==== Unigram Queries

Because the index contains unigrams,(((`"unigrams", "unigram phrase queries"`)))
(((`"common_grams token filter", "unigram queries"`))) the field can be queried using the same techniques that we have used for any other field, for example:

[source, json]

```
GET /my_index/_search { "query": { "match": { "text": { "query": "the quick and brown fox", "cutoff_frequency": 0.01 } } }}
```



The preceding query string is analyzed by the `search_analyzer` configured for the `text` field--the `standard` analyzer in this example--to produce the terms `the`, `quick`, `and`, `brown`, `fox`.

Because the index for the `text` field contains the same unigrams as produced by the `standard` analyzer, search functions as it would for any normal field.

==== Bigram Phrase Queries

However, when we come to do phrase queries,(((`"common_grams` token filter", "bigram phrase queries")))((`"bigrams`", "bigram phrase queries"))) we can use the specialized `search_grams` analyzer to make the process much more efficient:

[source,json]

```
GET /my_index/_search { "query": { "match_phrase": { "text": { "query": "The quick and brown fox", "analyzer": "search_grams" <1> } } } }
```

<1> For phrase queries, we override the default `search_analyzer` and use the `search_grams` analyzer instead.

The `search_grams` analyzer would produce the following terms:

[source,text]

Pos 1: `the_quick` Pos 2: `quick_and` Pos 3: `and_brown` Pos 4: `brown`

Pos 5: fox

The analyzer has stripped out all of the common word unigrams, leaving the common word bigrams and the low-frequency unigrams. Bigrams like `the_quick` are much less common than the single term `the`. This has two advantages:

- The positions data for `the_quick` is much smaller than for `the`, so it is faster to read from disk and has less of an impact on the filesystem cache.



- The term `the_quick` is much less common than `the`, so it drastically decreases the number of documents that have to be examined.

===== Two-Word Phrases

There is one further optimization. (((`"common_grams` token filter", "two word phrases"))) By far the majority of phrase queries consist of only two words. If one of those words happens to be a common word, such as

[source,json]

```
GET /my_index/_search { "query": { "match_phrase": { "text": { "query": "The quick",  
"analyzer": "search_grams" } } } }
```

```
}
```

then the `search_grams` analyzer outputs a single token: `the_quick`. This transforms what originally could have been an expensive phrase query for `the` and `quick` into a very efficient single-term lookup.



[[stopwords-relavance]] === Stopwords and Relevance

The last topic to cover before moving on from stopwords((("stopwords", "relevance and"))) ((("relevance", "stopwords and"))) is that of relevance. Leaving stopwords in your index could make the relevance calculation less accurate, especially if your documents are very long.

As we have already discussed in <>, the((("BM25", "term frequency saturation"))) reason for this is that <> doesn't impose an upper limit on the impact of term frequency.((("Term Frequency/Inverse Document Frequency (TF/IDF) similarity algorithm", "stopwords and"))) Very common words may have a low weight because of inverse document frequency but, in long documents, the sheer number of occurrences of stopwords in a single document may lead to their weight being artificially boosted.

You may want to consider using the <> similarity on long fields that include stopwords instead of the default Lucene similarity.



[[synonyms]] == Synonyms

While stemming helps to broaden the scope of search by simplifying inflected words to their root form, `synonyms(("synonyms"))` broaden the scope by relating concepts and ideas.

Perhaps no documents match a query for `English queen,` '' but documents that contain `British monarch`" would probably be considered a good match.

A user might search for `'the us'` and expect to find documents that contain `_United States_, _USA_, _U.S.A._, _America_, or _the States_`. However, they wouldn't expect to see results about `the states of matter or state machines`.

This example provides a valuable lesson. It demonstrates how simple it is for a human to distinguish between separate concepts, and how tricky it can be for mere machines. The natural tendency is to try to provide synonyms for every word in the language, to ensure that any document is findable with even the most remotely related terms.

This is a mistake. In the same way that we prefer light or minimal stemming to aggressive stemming, synonyms should be used only where necessary. Users understand why their results are limited to the words in their search query. They are less understanding when their results seems almost random.

Synonyms can be used to conflate words that have pretty much the same meaning, such as `jump , leap , and hop , or pamphlet , leaflet , and brochure`. Alternatively, they can be used to make a word more generic. For instance, `bird` could be used as a more general synonym for `owl` or `pigeon`, and `adult` could be used for `man` or `woman`.

Synonyms appear to be a simple concept but they are quite tricky to get right. In this chapter, we explain the mechanics of using synonyms and discuss the limitations and gotchas.

[TIP]

Synonyms are used to broaden the scope of what is considered a matching document. Just as with `<>` or `><`, synonym fields should not be used alone but should be combined with a query on a main field that contains the original text in unadulterated form. See `<>` for an

explanation of how to maintain relevance when using synonyms.



[[using-synonyms]] === Using Synonyms

Synonyms can replace existing tokens or(("synonyms", "using")) be added to the token stream by using the(("synonym token filter")) <http://bit.ly/1DInEGD>[`synonym` token filter]:

[source,json]

```
PUT /my_index { "settings": { "analysis": { "filter": { "my_synonym_filter": { "type": "synonym", <1> "synonyms": [ <2> "british,english", "queen,monarch" ] } }, "analyzer": { "my_synonyms": { "tokenizer": "standard", "filter": [ "lowercase", "my_synonym_filter" <3> ] } } } }
```

```
}
```

<1> First, we define a token filter of type `synonym`.

<2> We discuss synonym formats in <>.

<3> Then we create a custom analyzer that uses the `my_synonym_filter`.

[TIP]

Synonyms can be specified inline with the `synonyms` parameter, or in a `synonyms` file that must(("synonyms", "specifying inline or in a separate file")) be present on every node in the cluster. The path to the `synonyms` file should be specified with the `synonyms_path` parameter, and should be either absolute or relative to the Elasticsearch `config` directory. See <> for techniques that can be used to refresh the `synonyms` list.

=====

Testing our analyzer with the `analyze` API shows the following:

[source,json]

```
GET /my_index/_analyze?analyzer=my_synonyms
```

Elizabeth is the English queen

[source,text]



Pos 1: (elizabeth) Pos 2: (is) Pos 3: (the) Pos 4: (british,english) <1>

Pos 5: (queen,monarch) <1>

<1> All synonyms occupy the same position as the original term.

A document like this will match queries for any of the following: English queen , British queen , English monarch , or British monarch . Even a phrase query will work, because the position of each term has been preserved.

[TIP]

Using the same synonym token filter at both index time and search time is redundant. ((("synonym token filter", "using at index time versus search time"))) If, at index time, we replace English with the two terms english and british , then at search time we need to search for only one of those terms. Alternatively, if we don't use synonyms at index time, then at search time, we would need to convert a query for English into a query for english OR british .

Whether to do synonym expansion at search or index time can be a difficult choice. We will explore the options more in <>.

=====



[[synonym-formats]] === Formatting Synonyms

In their simplest form, synonyms are(("synonyms", "formatting")) listed as comma-separated values:

```
"jump,leap,hop"
```

If any of these terms is encountered, it is replaced by all of the listed synonyms. For instance:

```
[role="pagebreak-before"]
```

[source,text]

Original terms: Replaced by: _____ jump →
(jump,leap,hop) leap → (jump,leap,hop)

hop → (jump,leap,hop)

Alternatively, with the => syntax, it is possible to specify a list of terms to match (on the left side), and a list of one or more replacements (on the right side):

```
"u s a,united states,united states of america => usa"  
"g b,gb,great britain => britain,england,scotland,wales"
```

[source,text]

Original terms: Replaced by: _____ u s a →
(usa) united states → (usa)

great britain → (britain,england,scotland,wales)

If multiple rules for the same synonyms are specified, they are merged together. The order of rules is not respected. Instead, the longest matching rule wins. Take the following rules as an example:

```
"united states          => usa",  
"united states of america => usa"
```



If these rules conflicted, Elasticsearch would turn `United States of America` into the terms `(usa), (of), (america)`. Instead, the longest sequence wins, and we end up with just the term `(usa)`.



[[synonyms-expand-or-contract]] === Expand or contract

In <>, we have seen that it is((("synonyms", "expanding or contracting"))) possible to replace synonyms by *simple expansion*, *simple contraction*, or *generic expansion*. We will look at the trade-offs of each of these techniques in this section.

TIP: This section deals with single-word synonyms only. Multiword synonyms add another layer of complexity and are discussed later in <>.

[[synonyms-expansion]] === Simple Expansion

With *simple expansion*,((("synonyms", "expanding or contracting", "simple expansion")))((("simple expansion (synonyms)"))) any of the listed synonyms is expanded into *all* of the listed synonyms:

```
"jump, hop, leap"
```

Expansion can be applied either at index time or at query time. Each has advantages (\uparrow) and disadvantages (\downarrow). When to use which comes down to performance versus flexibility.

[options="header",cols="h,d,d"]

|===== | | Index time | Query time

| Index size | \downarrow Bigger index because all synonyms must be indexed. | \uparrow Normal.

| Relevance | \downarrow All synonyms will have the same IDF (see <>), meaning that more commonly used words will have the same weight as less commonly used words. | \uparrow The IDF for each synonym will be correct.

| Performance | \uparrow A query needs to find only the single term specified in the query string. | \downarrow A query for a single term is rewritten to look up all synonyms, which decreases performance.

| Flexibility | \downarrow The synonym rules can't be changed for existing documents. For the new rules to have effect, existing documents have to be reindexed. | \uparrow Synonym rules can be updated without reindexing documents.

|=====

[[synonyms-contraction]] === Simple Contraction

Simple contraction maps a group of ((("synonyms", "expanding or contracting", "simple contraction")))((("simple contraction (synonyms)")))synonyms on the left side to a single value on the right side:

```
"leap, hop => jump"
```



It must be applied both at index time and at query time, to ensure that query terms are mapped to the same single value that exists in the index.

This approach has some advantages and some disadvantages compared to the simple expansion approach:

Index size::

- ↑ The index size is normal, as only a single term is indexed.

Relevance::

- ↓ The IDF for all terms is the same, so you can't distinguish between more commonly used words and less commonly used words.

Performance::

- ↑ A query needs to find only the single term that appears in the index.

Flexibility::

+

↑ New synonyms can be added to the left side of the rule and applied at query time. For instance, imagine that we wanted to add the word `bound` to the rule specified previously. The following rule would work for queries that contain `bound` or for newly added documents that contain `bound`:

```
"leap, hop, bound => jump"
```

But we could expand the effect to also take into account *existing* documents that contain `bound` by writing the rule as follows:

```
"leap, hop, bound => jump, bound"
```

When you reindex your documents, you could revert to the previous rule to gain the performance benefit of querying only a single term.

--

`[[synonyms-genres]] ===== Genre Expansion`

Genre expansion is quite different from simple(((`"synonyms", "expanding or contracting", "genre expansion"`))))((`"genre expansion (synonyms)"`))) contraction or expansion. Instead of treating all synonyms as equal, genre expansion widens the meaning of a term to be more



generic. Take these rules, for example:

```
"cat    => cat,pet",
"kitten => kitten,cat,pet",
"dog    => dog,pet"
"puppy  => puppy,dog,pet"
```

By applying genre expansion at index time:

- A query for `kitten` would find just documents about kittens.
- A query for `cat` would find documents about kittens and cats.
- A query for `pet` would find documents about kittens, cats, puppies, dogs, or pets.

Alternatively, by applying genre expansion at query time, a query for `kitten` would be expanded to return documents that mention kittens, cats, or pets specifically.

You could also have the best of both worlds by applying expansion at index time to ensure that the genres are present in the index. Then, at query time, you can choose to not apply synonyms (so that a query for `kitten` returns only documents about kittens) or to apply synonyms in order to match kittens, cats and pets (including the canine variety).

With the preceding example rules above, the IDF for `kitten` will be correct, while the IDF for `cat` and `pet` will be artificially deflated. However, this works in your favor--a genre-expanded query for `kitten OR cat OR pet` will rank documents with `kitten` highest, followed by documents with `cat`, and documents with `pet` would be right at the bottom.



[[synonyms-analysis-chain]] === Synonyms and The Analysis Chain

The example we (((“synonyms”, “and the analysis chain”))) showed in <>, used `u s a` as a synonym. Why did we use that instead of `U.S.A.`? The reason is that the `synonym` token filter sees only the terms that the previous token filter or tokenizer has emitted.(((“analysis”, “synonyms and the analysis chain”)))

Imagine that we have an analyzer that consists of the `standard` tokenizer, with the `lowercase` token filter followed by a `synonym` token filter. The analysis process for the text `U.S.A.` would look like this:

[source, text]

original string → "U.S.A." standard tokenizer → (U),(S),(A) lowercase token filter → (u),(s),(a)

synonym token filter → (usa)

If we had specified the synonym as `U.S.A.`, it would never match anything because, by the time `my_synonym_filter` sees the terms, the periods have been removed and the letters have been lowercased.

This is an important point to consider. What if we want to combine synonyms with stemming, so that `jumps`, `jumped`, `jump`, `leaps`, `leaped`, and `leap` are all indexed as the single term `jump`? We (((“stemming words”, “combining synonyms with”))) could place the synonyms filter before the stemmer and list all inflections:

```
"jumps, jumped, leap, leaps, leaped => jump"
```

But the more concise way would be to place the synonyms filter after the stemmer, and to list just the root words that would be emitted by the stemmer:

```
"leap => jump"
```

==== Case-Sensitive Synonyms

Normally, synonym filters are placed after the `lowercase` token filter and so all synonyms are (((“synonyms”, “and the analysis chain”, “case-sensitive synonyms”)))(((“case-sensitive synonyms”))) written in lowercase, but sometimes that can lead to odd conflations. For



instance, a `CAT` scan and a `cat` are quite different, as are `PET` (positron emmision tomography) and a `pet`. For that matter, the surname `Little` is distinct from the adjective `little` (although if a sentence starts with the adjective, it will be uppercased anyway).

If you need use case to distinguish between word senses, you will need to place your synonym filter before the `lowercase` filter. Of course, that means that your synonym rules would need to list all of the case variations that you want to match (for example, `Little,LITTLE,little`).

Instead of that, you could have two synonym filters: one to catch the case-sensitive synonyms and one for all the case-insentive synonyms. For instance, the case-sensitive rules could look like this:

```
"CAT,CAT scan      => cat_scan"  
"PET,PET scan      => pet_scan"  
"Johnny Little,J Little => johnny_little"  
"Johnny Small,J Small => johnny_small"
```

And the case-insentive rules could look like this:

```
"cat          => cat,pet"  
"dog          => dog,pet"  
"cat scan,cat_scan scan => cat_scan"  
"pet scan,pet_scan scan => pet_scan"  
"little,small"
```

The case-sensitive rules would `CAT` scan but would match only the `CAT` in `CAT` scan . For this reason, we have the odd-looking rule `cat_scan scan` in the case-insensitive list to catch bad replacements.

TIP: You can see how quickly it can get complicated. As always, the `analyze` API is your friend--use it to check that your analyzers are configured correctly. See <[>](#)



[[multi-word-synonyms]] === Multiword Synonyms and Phrase Queries

So far, synonyms appear to be quite straightforward. Unfortunately, this is where things start to go wrong.(((("synonyms", "multiword, and phrase queries"))))((("phrase matching", "multiword synonyms and")))) For <> to function correctly, Elasticsearch needs to know the position that each term occupies in the original text. Multiword synonyms can play havoc with term positions, especially when the injected synonyms are of differing lengths.

To demonstrate, we'll create a synonym token filter that uses this rule:

```
"usa,united states,u s a,united states of america"
```

[source,json]

```
PUT /my_index { "settings": { "analysis": { "filter": { "my_synonym_filter": { "type": "synonym", "synonyms": [ "usa,united states,u s a,united states of america" ] } }, "analyzer": { "my_synonyms": { "tokenizer": "standard", "filter": [ "lowercase", "my_synonym_filter" ] } } } } }
```

```
GET /my_index/_analyze?analyzer=my_synonyms&text=
```

The United States is wealthy

The tokens emitted by the `analyze` request look like this:

[source,text]

Pos 1: (the) Pos 2: (usa,united,u,united) Pos 3: (states,s,states) Pos 4: (is,a,of)

Pos 5: (wealthy,america)

If we were to index a document analyzed with synonyms as above, and then run a phrase query without synonyms, we'd have some surprising results. These phrases would not match:

- The usa is wealthy
- The united states of america is wealthy
- The U.S.A. is wealthy

However, these phrases would:



- United states is wealthy
- Usa states of wealthy
- The U.S. of wealthy
- U.S. is america

If we were to use synonyms at query time instead, we would see even more-bizarre matches. Look at the output of this `validate-query` request:

[source,json]

```
GET /my_index/_validate/query?explain { "query": { "match_phrase": { "text": { "query": "usa is wealthy", "analyzer": "my_synonyms" } } } }
```

```
}
```

The explanation is as follows:

```
"(usa united u united) (is states s states) (wealthy a of) america"
```

This would match documents containing `u is of america` but wouldn't match any document that didn't contain the term `america`.

[TIP]

Multiword synonyms (((("highlighting searches", "multiword synonyms and")))) affect highlighting in a similar way. A query for `USA` could end up returning a highlighted snippet such as: ``The *United States is wealthy*''.

=====

==== Use Simple Contraction for Phrase Queries

The way to avoid this mess is to use `<>` to inject a single(((("synonyms", "multiword, and phrase queries", "using simple contraction"))))((("phrase matching", "multiword synonyms and", "using simple contraction"))))((("simple contraction (synonyms)", "using for phrase queries")))) term that represents all synonyms, and to use the same synonym token filter at query time:

[source,json]



```
PUT /my_index { "settings": { "analysis": { "filter": { "my_synonym_filter": { "type": "synonym", "synonyms": [ "united states,u s a,united states of america=>usa" ] } }, "analyzer": { "my_synonyms": { "tokenizer": "standard", "filter": [ "lowercase", "my_synonym_filter" ] } } } }
```

```
GET /my_index/_analyze?analyzer=my_synonyms
```

The United States is wealthy

The result of the preceding `analyze` request looks much more sane:

[source, text]

Pos 1: (the) Pos 2: (usa) Pos 3: (is)

Pos 5: (wealthy)

And repeating the `validate-query` request that we made previously yields a simple, sane explanation:

```
"usa is wealthy"
```

The downside of this approach is that, by reducing `united states of america` down to the single term `usa`, you can't use the same field to find just the word `united` or `states`. You would need to use a separate field with a different analysis chain for that purpose.

==== Synonyms and the query_string Query

We have tried to avoid discussing the `query_string` query (((("query strings", "synonyms and")))((("synonyms", "multiword, and query string queries"))))because we don't recommend using it. In <>, we said that, because the `query_string` query supports a terse mini `search-syntax`, it could frequently lead to surprising results or even syntax errors.

One of the gotchas of this query involves multiword synonyms. To support its search-syntax, it has to parse the query string to recognize special operators like `AND`, `OR`, `+`, `-`, `field:`, and so forth. (See the full <http://bit.ly/151G5I1>[`query_string` syntax] here.)

As part of this parsing process, it breaks up the query string on whitespace, and passes each word that it finds to the relevant analyzer separately. This means that your synonym analyzer will never receive a multiword synonym. Instead of seeing `United States` as a single string, the analyzer will receive `United` and `States` separately.



Fortunately, the trustworthy `match` query supports no such syntax, and multiword synonyms will be passed to the analyzer in their entirety.



[[symbol-synonyms]] === Symbol Synonyms

The final part of this chapter is devoted to symbol synonyms, which are unlike the `synonyms(("symbol synonyms"))((("synonyms", "symbol")))` we have discussed until now. *Symbol synonyms* are string aliases used to represent symbols that would otherwise be removed during tokenization.

While most punctuation is seldom important for full-text search, character combinations like `emoticons(("emoticons"))` may be very significant, even changing the meaning of the text. Compare these:

[role="pagebreak-before"]

- I am thrilled to be at work on Sunday.
- I am thrilled to be at work on Sunday :(

The `standard` tokenizer would simply strip out the emoticon in the second sentence, conflating two sentences that have quite different intent.

We can use the <http://bit.ly/1ziua5n>[`mapping` character filter] to replace `emoticons(("mapping character filter", "replacing emoticons with symbol synonyms"))((("emoticons", "replacing with symbol synonyms")))` with symbol synonyms like `emoticon_happy` and `emoticon_sad` before the text is passed to the tokenizer:

[source,json]

```
PUT /my_index { "settings": { "analysis": { "char_filter": { "emoticons": { "type": "mapping", "mappings": [ <1> ":"=>emoticon_happy", ":(=>emoticon_sad" ] } }, "analyzer": { "my_emoticons": { "char_filter": "emoticons", "tokenizer": "standard", "filter": [ "lowercase" ] } } } }}
```

```
GET /my_index/_analyze?analyzer=my_emoticons
```

I am :) not :(<2>

<1> The `mappings` filter replaces the characters to the left of `=>` with those to the right.

<2> Emits tokens `i`, `am`, `emoticon_happy`, `not`, `emoticon_sad`.

It is unlikely that anybody would ever search for `emoticon_happy`, but ensuring that important symbols like emoticons are included in the index can be helpful when doing sentiment analysis. Of course, we could equally have used real words, like `happy` and `sad`.



TIP: The `mapping` character filter is useful for simple replacements of exact character sequences. ((("mapping character filter", "replacements of exact character sequences"))) For more-flexible pattern matching, you can use regular expressions with the <http://bit.ly/1DK4hgy>[`pattern_replace` character filter].



[[模糊-匹配]] == 打字错误 和 拼写错误

我们希望在结构化数据上的查询（如日期和价格）仅返回精确匹配的文档. (((("typoes and misspellings", "fuzzy matching"))))((("fuzzy matching")))) 然而，好的全文检索不应该有同样的限制. 相反，我们能拓宽网络以包含那些可能的匹配，并且利用相关性分数把更好的匹配结果放在结果集的前面.

事实上，仅能精确匹配的全文检索 ((("full text search", "fuzzy matching")))) 可能会让你的用户感到失望. 难道你不希望一个对 `quick brown fox'` 的检索能匹配包含 `fast brown foxes,'` 的文档，对 `Johnny Walker'` 的检索能匹配包含 `Johnnie Walker,'` 的文档 或 `Arnold Shcwazenneger'` 能匹配 `Arnold Schwarzenegger"` 吗？

如果文档中存在 确切 包含于用户查询的内容，它们应该出现在结果集的前面，但更弱的匹配可能在下面的列表中.

如果没有精确匹配的文档，至少我们应该为用户显示可能的匹配结果；它们甚至有可能就是用户原来想要的！

我们已经在 <> 看过 diacritic-free 匹配，在 <> 看过 词干提取，在 <> 看过 同义词，但是所有的这些方法都预先假定了单词是正确拼写的，或者每个词只有一种拼写方式.

模糊匹配允许 查询-时 匹配拼写错误的单词，音标表征过滤器能在索引时用于 发音-相似 的匹配.



[[模糊]] === 模糊

模糊匹配 视两个单词“模糊”相似，正好像它们是同一个词。((("typoes and misspellings", "fuzziness, defining")))) 首先，我们需要通过 **fuzziness** 来定义什么是((("fuzziness"))))。

1965年, Vladimir Levenshtein 开发了

http://en.wikipedia.org/wiki/Levenshtein_distance[Levenshtein distance(Levenshtein 距离)]

用来度量把一个单词转换为另一个单词需要的单字符编辑次数 ((("Levenshtein distance"))))。他提出了3种单字符编辑：

- 替换 一个字符到另一个字符: `_f_ox -> _b_ox`
- 插入 一个新字符: `sic -> sick`
- 删 除 一个字符: `b_l_ack -> back`

http://en.wikipedia.org/wiki/Frederick_J._Damerau[Frederick Damerau] 稍后扩展了这些操作并包含了1个新的 ((("Damerau, Frederick J."))):

- 换位 调整字符: `_st_ar -> _ts_ar`

例如, 把 `bieber` 转换为 `beaver` 需要以下几步:

1. 用 `v` 替换掉 `b`: `bie_b_er -> bie_v_er`
2. 用 `a` 替换掉 `i`: `b_i_ever -> b_a_ever`
3. 换位 `a` 和 `e`: `b_ae_ver -> b_ea_ver`

以上的3步代表了3个 <http://bit.ly/1ymgZPB>[Damerau-Levenshtein edit distance(Damerau-Levenshtein 编辑距离)]。

显然, `ieber` 距 `beaver` — 很远; 远得无法被认为是一个简单的拼写错误. Damerau发现 80% 的人类拼写错误的编辑距离都是1. 换句话说, 80% 的拼写错误都可以通过 单次编辑修改为原始的字符串.

通过指定 `fuzziness` 参数为 2,Elasticsearch 支持最大的编辑距离.

当然, 一个字符串的单次编辑次数依赖于它的长度. 对 `hat` 进行两次编辑可以得到 `mad`, 所以允许对长度为3的字符串进行两次修改就太过了. `fuzziness` 参数可以被设置成 `AUTO`, 结果会在下面的最大编辑距离中:

- 0 1或2个字符的字符串
- 1 3、4或5个字符的字符串
- 2 多于5个字符的字符串

当然, 你可能发现编辑距离为 2 仍然是太过了, 返回的结果好像并没有什么关联. 把 `fuzziness` 设置为 1 ,你可能会获得更好的结果和性能.



[[fuzzy-query]] === Fuzzy Query

The <http://bit.ly/1ymh8Cu> query is (((("typos and misspellings", "fuzzy query")))) (((("fuzzy queries")))) the fuzzy equivalent of the `term` query. You will seldom use it directly yourself, but understanding how it works will help you to use fuzziness in the higher-level `match` query.

To understand how it works, we will first index some documents:

[source,json]

```
POST /my_index/my_type/_bulk { "index": { "_id": 1 } } { "text": "Surprise me!" } { "index": { "_id": 2 } } { "text": "That was surprising." } { "index": { "_id": 3 } }
```

```
{ "text": "I wasn't surprised." }
```

Now we can run a `fuzzy` query for the term `surprise`:

[source,json]

```
GET /my_index/my_type/_search { "query": { "fuzzy": { "text": "surprise" } } }
```

```
}
```

The `fuzzy` query is a term-level query, so it doesn't do any analysis. It takes a single term and finds all terms in the term dictionary that are within the specified `fuzziness`. The default `fuzziness` is `AUTO`.

In our example, `surprise` is within an edit distance of 2 from both `surprise` and `surprised`, so documents 1 and 3 match. We could reduce the matches to just `surprise` with the following query:

[source,json]

```
GET /my_index/my_type/_search { "query": { "fuzzy": { "text": { "value": "surprise", "fuzziness": 1 } } } }
```



==== Improving Performance

The `fuzzy` query works by taking the original term and building a *Levenshtein automaton*—like a`((("fuzzy queries", "improving performance")))((("Levenshtein automation")))` big graph representing all the strings that are within the specified edit distance of the original string.

The fuzzy query then uses the automation to step efficiently through all of the terms in the term dictionary to see if they match. Once it has collected all of the matching terms that exist in the term dictionary, it can compute the list of matching documents.

Of course, depending on the type of data stored in the index, a fuzzy query with an edit distance of 2 can match a very large number of terms and perform very badly. Two parameters can be used to limit the performance impact:

`prefix_length ::`

The number of initial characters`((("prefix_length parameter")))` that will not be ``fuzzified.'` Most spelling errors occur toward the end of the word, not toward the beginning. By using a `prefix_length of 3`, for example, you can significantly reduce the number of matching terms.

`max_expansions ::`

If a fuzzy query expands to three or four fuzzy options,`((("max_expansions parameter")))` the new options may be meaningful. If it produces 1,000 options, they are essentially meaningless. Use `max_expansions` to limit the total number of options that will be produced. The fuzzy query will collect matching terms until it runs out of terms or reaches the `max_expansions` limit.



[[fuzzy-match-query]] === Fuzzy match Query

The `match` query supports ((("typoes and misspellings", "fuzzy match query")))((("match query", "fuzzy matching")))((("fuzzy matching", "match query")))fuzzy matching out of the box:

[source,json]

```
GET /my_index/my_type/_search { "query": { "match": { "text": { "query": "SURPRISE ME!", "fuzziness": "AUTO", "operator": "and" } } }}
```

}

The query string is first analyzed, to produce the terms `[surprise, me]`, and then each term is fuzzified using the specified `fuzziness`.

Similarly, the `multi_match` query also (((("multi_match queries", "fuzziness support"))))supports `fuzziness`, but only when executing with type `best_fields` or `most_fields`:

[source,json]

```
GET /my_index/my_type/_search { "query": { "multi_match": { "fields": [ "text", "title" ], "query": "SURPRISE ME!", "fuzziness": "AUTO" } }}
```

}

Both the `match` and `multi_match` queries also support the `prefix_length` and `max_expansions` parameters.

TIP: Fuzziness works only with the basic `match` and `multi_match` queries. It doesn't work with phrase matching, common terms, or `cross_fields` matches.



[[fuzzy-scoring]] === Scoring Fuzziness

Users love fuzzy queries. They assume that these queries will somehow magically find the right combination of proper spellings.((("fuzzy queries", "scoring fuzziness")))(("typos and misspellings", "scoring fuzziness")))(("relevance scores", "fuzziness and"))). Unfortunately, the truth is somewhat more prosaic.

Imagine that we have 1,000 documents containing Schwarzenegger, '' and just one document with the misspelling Schwarzeneger." According to the theory of <>, the misspelling is much more relevant than the correct spelling, because it appears in far fewer documents!

In other words, if we were to treat fuzzy matches((("match query", "fuzzy match query"))) like any other match, we would favor misspellings over correct spellings, which would make for grumpy users.

TIP: Fuzzy matching should not be used for scoring purposes--only to widen the net of matching terms in case there are misspellings.

By default, the `match` query gives all fuzzy matches the constant score of 1. This is sufficient to add potential matches onto the end of the result list, without interfering with the relevance scoring of nonfuzzy queries.

[TIP]

Fuzzy queries alone are much less useful than they initially appear. They are better used as part of a `bigger' feature, such as the `_search-as-you-type` <http://bit.ly/1IChV5j> [`completion` suggester] or the `_did-you-mean` <http://bit.ly/1I0j5ZG> [`phrase`` suggester].

=====



[[phonetic-matching]] === Phonetic Matching

In a last, desperate, attempt to match something, anything, we could resort to searching for words that sound similar, (((("typos and misspellings", "phonetic matching"))))((("phonetic matching"))))even if their spelling differs.

Several algorithms exist for converting words into a phonetic representation.((("phonetic algorithms"))) The [http://en.wikipedia.org/wiki/Soundex\[Soundex\]](http://en.wikipedia.org/wiki/Soundex[Soundex]) algorithm is the granddaddy of them all, and most other phonetic algorithms are improvements or specializations of Soundex, such as [http://en.wikipedia.org/wiki/Metaphone\[Metaphone\]](http://en.wikipedia.org/wiki/Metaphone[Metaphone]) and [http://en.wikipedia.org/wiki/Metaphone#Double_Metaphone\[Double Metaphone\]](http://en.wikipedia.org/wiki/Metaphone#Double_Metaphone[Double Metaphone]) (which expands phonetic matching to languages other than English), [http://en.wikipedia.org/wiki/Caverphone\[Caverphone\]](http://en.wikipedia.org/wiki/Caverphone[Caverphone]) for matching names in New Zealand, the [http://bit.ly/1E47qoB\[Beider-Morse\]](http://bit.ly/1E47qoB[Beider-Morse]) algorithm, which adopts the Soundex algorithm for better matching of German and Yiddish names, and the [http://de.wikipedia.org/wiki/K%C3%B6lner_Phonetik\[K%C3%B6lner Phonetik\]](http://de.wikipedia.org/wiki/K%C3%B6lner_Phonetik[K%C3%B6lner Phonetik]) for better handling of German words.

The thing to take away from this list is that phonetic algorithms are fairly crude, and (((("languages", "phonetic algorithms"))))very specific to the languages they were designed for, usually either English or German. This limits their usefulness. Still, for certain purposes, and in combination with other techniques, phonetic matching can be a useful tool.

First, you will need to install ((("Phonetic Analysis plugin")))the Phonetic Analysis plug-in from <http://bit.ly/1CreKJQ> on every node in the cluster, and restart each node.

Then, you can create a custom analyzer that uses one of the phonetic token filters ((("phonetic matching", "creating a phonetic analyzer"))))and try it out:

[source,json]

```
PUT /my_index { "settings": { "analysis": { "filter": { "dbl_metaphone": { <1> "type": "phonetic", "encoder": "double_metaphone" } }, "analyzer": { "dbl_metaphone": { "tokenizer": "standard", "filter": "dbl_metaphone" <2> } } } }
```

```
}
```

<1> First, configure a custom `phonetic` token filter that uses the `double_metaphone` encoder.

<2> Then use the custom token filter in a custom analyzer.



Now we can test it with the `analyze` API:

[source,json]

```
GET /my_index/_analyze?analyzer=dbl_metaphone
```

Smith Smythe

Each of `Smith` and `Smythe` produce two tokens in the same position: `SM0` and `XMT`. Running `John`, `Jon`, and `Johnnie` through the analyzer will all produce the two tokens `JN` and `AN`, while `Jonathon` results in the tokens `JN0N` and `ANTN`.

The phonetic analyzer can be used just like any other analyzer. First map a field to use it, and then index some data:

[source,json]

```
PUT /my_index/_mapping/my_type { "properties": { "name": { "type": "string", "fields": { "phonetic": { <1> "type": "string", "analyzer": "dbl_metaphone" } } } }}
```

```
PUT /my_index/my_type/1 { "name": "John Smith" }
```

```
PUT /my_index/my_type/2 { "name": "Jonnie Smythe" }
```

```
}
```

<1> The `name.phonetic` field uses the custom `dbl_metaphone` analyzer.

The `match` query can be used for searching:

[source,json]

```
GET /my_index/my_type/_search { "query": { "match": { "name.phonetic": { "query": "Jahnnie Smeeth", "operator": "and" } } }}
```

```
}
```



This query returns both documents, demonstrating just how coarse phonetic matching is.
(("phonetic matching", "purpose of")) Scoring with a phonetic algorithm is pretty much worthless. The purpose of phonetic matching is not to increase precision, but to increase recall--to spread the net wide enough to catch any documents that might possibly match.
(("recall", "increasing with phonetic matching"))

It usually makes more sense to use phonetic algorithms when retrieving results which will be consumed and post-processed by another computer, rather than by human users.



[[aggregations]] = Aggregations

[partintro]

Up until this point, this book has been dedicated to search. With search, we have a query and we wish to find a subset of documents which match the query. We are looking for the proverbial needle(s) in the haystack.

With aggregations, we zoom out to get an overview of our data. Instead of looking for individual documents, we want to analyze and summarize our complete set of data.

// Popular manufacturers? Unusual clumps of needles in the haystack?

- How many needles are in the haystack?
- What is the average length of the needles?
- What is the median length of needle broken down by manufacturer?
- How many needles were added to the haystack each month?

Aggregations can answer more subtle questions too, such as

- What are your most popular needle manufacturers?
- Are there any unusual or anomalous clumps of needles?

Aggregations allow us to ask sophisticated questions of our data. And yet, while the functionality is completely different from search, it leverages the same data-structures. This means aggregations execute quickly and are *near-realtime*, just like search.

This is extremely powerful for reporting and dashboards. Instead of performing "rollups" of your data (e.g. *that crusty hadoop job that takes a week to run*), you can visualize your data in realtime, allowing you to respond immediately.

// Perhaps mention "not precalculated, out of date, and irrelevant"? // Perhaps "aggs are calculated in the context of the user's search, so you're not showing them that you have 10 4 star hotels on your site, but that you have 10 4 star hotels that *match their criteria*". Finally, aggregations operate alongside search requests. This means you can both search/filter documents *and* perform analytics at the same time, on the same data, in a single request. And because aggregations are calculated in the context of a user's search, you're not just displaying a count of four-star hotels... you're displaying a count of four-star hotels that *match their search criteria*.

Aggregations are so powerful that many companies have built large Elasticsearch

clusters solely for analytics.





==== Limiting Memory Usage

In order for aggregations (or any operation that requires access to field values) to be fast, (((("aggregations", "limiting memory usage"))))access to fielddata must be fast, which is why it is loaded into memory. (((("fielddata"))))(((("memory usage", "limiting for aggregations", "id="ix_memagg")))) But loading too much data into memory will cause slow garbage collections as the JVM tries to find extra space in the heap, or possibly even an OutOfMemory exception.

It may surprise you to find that Elasticsearch does not load into fielddata just the values for the documents that match your query. It loads the values for *all documents in your index*, even documents with a different `_type` !

The logic is: if you need access to documents X, Y, and Z for this query, you will probably need access to other documents in the next query. It is cheaper to load all values once, and to *keep them in memory*, than to have to scan the inverted index on every request.

The JVM heap (((("JVM (Java Virtual Machine)", "heap usage, fielddata and"))))is a limited resource that should be used wisely. A number of mechanisms exist to limit the impact of fielddata on heap usage. These limits are important because abuse of the heap will cause node instability (thanks to slow garbage collections) or even node death (with an OutOfMemory exception).

.Choosing a Heap Size

There are two rules to apply when setting (((("heap", rules for setting size of"))))the Elasticsearch heap size, with the `$ES_HEAP_SIZE` environment variable:

No more than 50% of available RAM:: Lucene makes good use of the filesystem caches, which are managed by the kernel. Without enough filesystem cache space, performance will suffer.

No more than 32 GB: If the heap is less than 32 GB, the JVM can use compressed pointers, which saves a lot of memory: 4 bytes per pointer instead of 8 bytes. + Increasing the heap from 32 GB to 34 GB would mean that you have much *less* memory available, because all pointers are taking double the space. Also, with bigger heaps, garbage collection becomes more costly and can result in node instability.

This limit has a direct impact on the amount of memory that can be devoted to fielddata.

[[fielddata-size]] ===== Fielddata Size



The `indices.fielddata.cache.size` controls how much heap space is allocated to fielddata.
((("fielddata", "size")))(("aggregations", "limiting memory usage", "fielddata size")) When you run a query that requires access to new field values, it will load the values into memory and then try to add them to fielddata. If the resulting fielddata size would exceed the specified `size`, other values would be evicted in order to make space.

By default, this setting is *unbounded*—Elasticsearch will never evict data from fielddata.

This default was chosen deliberately: fielddata is not a transient cache. It is an in-memory data structure that must be accessible for fast execution, and it is expensive to build. If you have to reload data for every request, performance is going to be awful.

A bounded size forces the data structure to evict data. We will look at when to set this value, but first a warning:

[WARNING]

This setting is a safeguard, not a solution for insufficient memory.

If you don't have enough memory to keep your fielddata resident in memory, Elasticsearch will constantly have to reload data from disk, and evict other data to make space. Evictions cause heavy disk I/O and generate a large amount of garbage in memory, which must be garbage collected later on.

=====

Imagine that you are indexing logs, using a new index every day. Normally you are interested in data from only the last day or two. Although you keep older indices around, you seldom need to query them. However, with the default settings, the fielddata from the old indices is never evicted! fielddata will just keep on growing until you trip the fielddata circuit breaker (see <>), which will prevent you from loading any more fielddata.

At that point, you're stuck. While you can still run queries that access fielddata from the old indices, you can't load any new values. Instead, we should evict old values to make space for the new values.

To prevent this scenario, place an upper limit on the fielddata by adding this setting to the `config/elasticsearch.yml` file:

[source,yaml]



indices.fielddata.cache.size: 40% <1>

<1> Can be set to a percentage of the heap size, or a concrete value like 5gb

With this setting in place, the least recently used fielddata will be evicted to make space for newly loaded data.((("fielddata", "expiry")))

[WARNING]

There is another setting that you may see online: indices.fielddata.cache.expire .

We beg that you *never* use this setting! It will likely be deprecated in the future.

This setting tells Elasticsearch to evict values from fielddata if they are older than expire , whether the values are being used or not.

This is *terrible* for performance. Evictions are costly, and this effectively *schedules* evictions on purpose, for no real gain.

There isn't a good reason to use this setting; we literally cannot theory-craft a hypothetically useful situation. It exists only for backward compatibility at the moment. We mention the setting in this book only since, sadly, it has been recommended in various articles on the Internet as a good performance tip.

It is not. Never use it!

[[monitoring-fielddata]] ===== Monitoring fielddata

It is important to keep a close watch on how much memory((("fielddata", "monitoring"))) (((("aggregations", "limiting memory usage", "moitoring fielddata")))) is being used by fielddata, and whether any data is being evicted. High eviction counts can indicate a serious resource issue and a reason for poor performance.

Fielddata usage can be monitored:

- per-index using the <http://bit.ly/1BwZ61b>[`indices-stats` API]: + [source,json]

GET /_stats/fielddata?fields=*

- per-node using the <http://bit.ly/1586yDn>[`nodes-stats` API]: + [source,json]



GET /_nodes/stats/indices/fielddata?fields=*

- Or even per-index per-node:

[source,json]

GET /_nodes/stats/indices/fielddata? level=indices&fields=*

By setting `?fields=*`, the memory usage is broken down for each field.

`[[circuit-breaker]]` ===== Circuit Breaker

An astute reader might have noticed a problem with the fielddata size settings. fielddata size is checked *after* the data is loaded.((("aggregations", "limiting memory usage", "fielddata circuit breaker"))) What happens if a query arrives that tries to load more into fielddata than available memory? The answer is ugly: you would get an OutOfMemoryException.
((("OutOfMemoryException")))(("circuit breakers")))

Elasticsearch includes a *fielddata circuit breaker* that is designed to deal with this situation. ((("fielddata circuit breaker"))) The circuit breaker estimates the memory requirements of a query by introspecting the fields involved (their type, cardinality, size, and so forth). It then checks to see whether loading the required fielddata would push the total fielddata size over the configured percentage of the heap.

If the estimated query size is larger than the limit, the circuit breaker is *tripped* and the query will be aborted and return an exception. This happens *before* data is loaded, which means that you won't hit an OutOfMemoryException.

.Available Circuit Breakers

Elasticsearch has a family of circuit breakers, all of which work to ensure that memory limits are not exceeded:

```
indices.breaker.fielddata.limit ::
```

The `fielddata` circuit breaker limits the size of fielddata to 60% of the heap, by default.



```
indices.breaker.request.limit ::
```

The `request` circuit breaker estimates the size of structures required to complete other parts of a request, such as creating aggregation buckets, and limits them to 40% of the heap, by default.

```
indices.breaker.total.limit ::
```

The `total` circuit breaker wraps the `request` and `fielddata` circuit breakers to ensure that the combination of the two doesn't use more than 70% of the heap by default.

The circuit breaker limits can be specified in the `config/elasticsearch.yml` file, or can be updated dynamically on a live cluster:

[source,js]

```
PUT /_cluster/settings { "persistent" : { "indices.breaker.fielddata.limit" : "40%" <1> } }
```

```
}
```

<1> The limit is a percentage of the heap.

It is best to configure the circuit breaker with a relatively conservative value. Remember that fielddata needs to share the heap with the `request` circuit breaker, the indexing memory buffer, the filter cache, Lucene data structures for open indices, and various other transient data structures. For this reason, it defaults to a fairly conservative 60%. Overly optimistic settings can cause potential OOM exceptions, which will take down an entire node.

On the other hand, an overly conservative value will simply return a query exception that can be handled by your application. An exception is better than a crash. These exceptions should also encourage you to reassess your query: why does a single query need more than 60% of the heap?

[TIP]



In <>, we spoke about adding a limit to the size of `fielddata`, to ensure that old unused `fielddata` can be evicted. The relationship between `indices.fielddata.cache.size` and `indices.breaker.fielddata.limit` is an important one. If the circuit-breaker limit is lower than the cache size, no data will ever be evicted. In order for it to work properly, the circuit breaker limit *must* be higher than the cache size.

=====

It is important to note that the circuit breaker compares estimated query size against the total heap size, *not* against the actual amount of heap memory used. This is done for a variety of technical reasons (for example, the heap may look full but is actually just garbage waiting to be collected, which is hard to estimate properly). But as the end user, this means the setting needs to be conservative, since it is comparing against total heap, not *free* heap. (((`"memory usage", "limiting for aggregations", startref = "ix_memagg"`)))



== Fielddata Filtering

Imagine that you are running a website that allows users to listen to their favorite songs. (((("fielddata", "filtering"))))(((("aggregations", "fielddata", "filtering")))) To make it easier for them to manage their music library, users can tag songs with whatever tags make sense to them. You will end up with a lot of tracks tagged with `rock`, `hiphop`, and `electronica`, but also with some tracks tagged with `my_16th_birthday_favorite_anthem`.

Now imagine that you want to show users the most popular three tags for each song. It is highly likely that tags like `rock` will show up in the top three, but `my_16th_birthday_favorite_anthem` is very unlikely to make the grade. However, in order to calculate the most popular tags, you have been forced to load all of these one-off terms into memory.

Thanks to fielddata filtering, we can take control of this situation. We *know* that we're interested in only the most popular terms, so we can simply avoid loading any terms that fall into the less interesting long tail:

[source,js]

```
PUT /music/_mapping/song { "properties": { "tag": { "type": "string", "fielddata": { <1> "filter": { "frequency": { <2> "min": 0.01, <3> "min_segment_size": 500 <4> } } } } }
```

```
}
```

<1> The `fielddata` key allows us to configure how fielddata is handled for this field.

<2> The `frequency` filter allows us to filter fielddata loading based on term frequencies. (((("term frequency", "fielddata filtering based on"))))

<3> Load only terms that occur in at least 1% of documents in this segment.

<4> Ignore any segments that have fewer than 500 documents.

With this mapping in place, only terms that appear in at least 1% of the documents *in that segment* will be loaded into memory. You can also specify a `max` term frequency, which could be used to exclude terms that are *too* common, such as <>.

Term frequencies, in this case, are calculated per segment. This is a limitation of the implementation: fielddata is loaded per segment, and at that point the only term frequencies that are visible are the frequencies for that segment. However, this limitation has interesting properties: it allows newly popular terms to rise to the top quickly.



Let's say that a new genre of song becomes popular one day. You would like to include the tag for this new genre in the most popular list, but if you were relying on term frequencies calculated across the whole index, you would have to wait for the new tag to become as popular as `rock` and `electronica`. Because of the way frequency filtering is implemented, the newly added tag will quickly show up as a high-frequency tag within new segments, so will quickly float to the top.

The `min_segment_size` parameter tells Elasticsearch to ignore segments below a certain size.((("min_segment_size parameter"))) If a segment holds only a few documents, the term frequencies are too coarse to have any meaning. Small segments will soon be merged into bigger segments, which will then be big enough to take into account.

[TIP]

Filtering terms by frequency is not the only option. You can also decide to load only those terms that match a regular expression. For instance, you could use a `regex` filter ((("regex filtering"))on tweets to load only hashtags into memory -- terms that start with a `#`. This assumes that you are using an analyzer that

preserves punctuation, like the whitespace analyzer.

Fielddata filtering can have a *massive* impact on memory usage. The trade-off is fairly obvious: you are essentially ignoring data. But for many applications, the trade-off is reasonable since the data is not being used anyway. The memory savings is often more important than including a large and relatively useless long tail of terms.



==== What about Facets?

If you've used Elasticsearch in the past, you are probably aware of *facets*. You can think of Aggregations as "facets on steroids". Everything you can do with facets, you can do with aggregations.

But there are plenty of operations that are possible in aggregations which are simply impossible with facets.

Facets have not been officially deprecated yet, but you can expect that to happen eventually. We recommend migrating your facets over to aggregations when you get the chance, and starting all new projects with aggregations instead of facets.



[[doc-values]] === Doc Values

In-memory fielddata is limited by the size of your heap.((("aggregations", "doc values"))) While this is a problem that can be solved by scaling horizontally--you can always add more nodes--you will find that heavy use of aggregations and sorting can exhaust your heap space while other resources on the node are underutilized.

While fielddata defaults to loading values into memory on the fly, this is not the only option. It can also be written to disk at index time in a way that provides all the functionality of in-memory fielddata, but without the heap memory usage. This alternative format is ((("fielddata", "doc values")))((("doc values"))) called *doc values*.

Doc values were added to Elasticsearch in version 1.0.0 but, until recently, they were much slower than in-memory fielddata. By benchmarking and profiling performance, various bottlenecks have been identified--in both Elasticsearch and Lucene--and removed.

Doc values are now only about 10–25% slower than in-memory fielddata, and come with two major advantages:

- They live on disk instead of in heap memory. This allows you to work with quantities of fielddata that would normally be too large to fit into memory. In fact, your heap space (`$ES_HEAP_SIZE`) can now be set to a smaller size, which improves the speed of garbage collection and, consequently, node stability.
- Doc values are built at index time, not at search time. While in-memory fielddata has to be built on the fly at search time by uninverting the inverted index, doc values are prebuilt and much faster to initialize.

The trade-off is a larger index size and slightly slower fielddata access. Doc values are remarkably efficient, so for many queries you might not even notice the slightly slower speed. Combine that with faster garbage collections and improved initialization times and you may notice a net gain.

The more filesystem cache space that you have available, the better doc values will perform. If the files holding the doc values are resident in the filesystem cache, then accessing the files is almost equivalent to reading from RAM. And the filesystem cache is managed by the kernel instead of the JVM.

===== Enabling Doc Values

Doc values can be enabled for numeric, date, Boolean, binary, and geo-point fields, and for `not_analyzed` string fields.((("doc values", "enabling"))) They do not currently work with `analyzed` string fields. Doc values are enabled per field in the field mapping, which means that you can combine in-memory fielddata with doc values:



[source,js]

```
PUT /music/_mapping/song { "properties" : { "tag": { "type": "string", "index" : "not_analyzed",  
"doc_values": true <1> } } }
```

}

<1> Setting `doc_values` to `true` at field creation time is all that is required to use disk-based fielddata instead of in-memory fielddata.

That's it! Queries, aggregations, sorting, and scripts will function as normal; they'll just be using doc values now. There is no other configuration necessary.

[TIP]

Use doc values freely. The more you use them, the less stress you place on the heap. It is possible that doc values will become the default format in the near future.

=====



[[preload-fielddata]] === Preloading Fielddata

The default behavior of Elasticsearch is to ((({"fielddata", "pre-loading"})))load in-memory fielddata *lazily*. The first time Elasticsearch encounters a query that needs fielddata for a particular field, it will load that entire field into memory for each segment in the index.

For small segments, this requires a negligible amount of time. But if you have a few 5 GB segments and need to load 10 GB of fielddata into memory, this process could take tens of seconds. Users accustomed to subsecond response times would all of a sudden be hit by an apparently unresponsive website.

There are three methods to combat this latency spike:

- Eagerly load fielddata
- Eagerly load global ordinals
- Prepopulate caches with warmers

All are variations on the same concept: preload the fielddata so that there is no latency spike when the user needs to execute a search.

[[eager-fielddata]] === Eagerly Loading Fielddata

The first tool is called *eager loading* (as opposed ((({"eager loading", "of fielddata"})))to the default lazy loading). As new segments are created (by refreshing, flushing, or merging), fields with eager loading enabled will have their per-segment fielddata preloaded *before* the segment becomes visible to search.

This means that the first query to hit the segment will not need to trigger fielddata loading, as the in-memory cache has already been populated. This prevents your users from experiencing the *cold cache* latency spike.

Eager loading is enabled on a per-field basis, so you can control which fields are pre-loaded:

[source,js]

```
PUT /music/_mapping/_song { "price_usd": { "type": "integer", "fielddata": { "loading": "eager" <1> } } }
```

```
}
```

<1> By setting `fielddata.loading: eager`, we tell Elasticsearch to preload this field's contents into memory.



Fielddata loading can be set to `lazy` or `eager` on existing fields, using the `update-mapping` API.

[WARNING]

Eager loading simply shifts the cost of loading fielddata. Instead of paying at query time, you pay at refresh time.

Large segments will take longer to refresh than small segments. Usually, large segments are created by merging smaller segments that are already visible to search, so the slower refresh time is not important.

=====

[[global-ordinals]] ===== Global Ordinals

One of the techniques used to reduce the memory usage of string fielddata is `(("ordinals"))` called *ordinals*.

Imagine that we have a billion documents, each of which has a `status` field. There are only three statuses: `status_deleted`, `status_pending`, `status_published`. If we were to hold the full string status in memory for every document, we would use 14 to 16 bytes per document, or about 15 GB.

Instead, we can identify the three unique strings, sort them, and number them: 0, 1, 2.

Ordinal		Term

0		<code>status_deleted</code>
1		<code>status_pending</code>
2		<code>status_published</code>

The original strings are stored only once in the ordinals list, and each document just uses the numbered ordinal to point to the value that it contains.

Doc		Ordinal

0		1 # pending
1		1 # pending
2		2 # published
3		0 # deleted

This reduces memory usage from 15 GB to less than 1 GB!



But there is a problem. Remember that fielddata caches are *per segment*. If one segment contains only two statuses—`status_deleted` and `status_published`—then the resulting ordinals (0 and 1) will not be the same as the ordinals for a segment that contains all three statuses.

If we try to run a `terms` aggregation on the `status` field, we need to aggregate on the actual string values, which means that we need to identify the same values across all segments. A naive way of doing this would be to run the aggregation on each segment, return the string values from each segment, and then reduce them into an overall result. While this would work, it would be slow and CPU intensive.

Instead, we use a structure called *global ordinals*. (((“global ordinals”))) Global ordinals are a small in-memory data structure built on top of fielddata. Unique values are identified across *all segments* and stored in an ordinals list like the one we have already described.

Now, our `terms` aggregation can just aggregate on the global ordinals, and the conversion from ordinal to actual string value happens only once at the end of the aggregation. This increases performance of aggregations (and sorting) by a factor of three or four.

===== Building global ordinals

Of course, nothing in life is free. (((“global ordinals”, “building”))) Global ordinals cross all segments in an index, so if a new segment is added or an old segment is deleted, the global ordinals need to be rebuilt. Rebuilding requires reading every unique term in every segment. The higher the cardinality--the more unique terms that exist--the longer this process takes.

Global ordinals are built on top of in-memory fielddata and doc values. In fact, they are one of the major reasons that doc values perform as well as they do.

Like fielddata loading, global ordinals are built lazily, by default. The first request that requires fielddata to hit an index will trigger the building of global ordinals. Depending on the cardinality of the field, this can result in a significant latency spike for your users. Once global ordinals have been rebuilt, they will be reused until the segments in the index change: after a refresh, a flush, or a merge.

[[eager-global-ordinals]] ===== Eager global ordinals

Individual string fields(((“eager loading”, “of global ordinals”)))(((“global ordinals”, “eager”))) can be configured to prebuild global ordinals eagerly:

[source,js]

```
PUT /music/_mapping/_song { "song_title": { "type": "string", "fielddata": { "loading" : "eager_global_ordinals" <1> } }
```



<1> Setting `eager_global_ordinals` also implies loading `fielddata` eagerly.

Just like the eager preloading of `fielddata`, eager global ordinals are built before a new segment becomes visible to search.

[NOTE]

Ordinals are only built and used for strings. Numerical data (integers, geopoints, dates, etc) doesn't need an ordinal mapping, since the value itself acts as an intrinsic ordinal mapping.

Therefore, you can only enable eager global ordinals for string fields.

Doc values can also have their global ordinals built eagerly:

[source,js]

```
PUT /music/_mapping/_song { "song_title": { "type": "string", "doc_values": true, "fielddata": {  
"loading" : "eager_global_ordinals" <1> } } }
```

}

<1> In this case, `fielddata` is not loaded into memory, but doc values are loaded into the filesystem cache.

Unlike `fielddata` preloading, eager building of global ordinals can have an impact on the *real-time* aspect of your data. For very high cardinality fields, building global ordinals can delay a refresh by several seconds. The choice is between paying the cost on each refresh, or on the first query after a refresh. If you index often and query seldom, it is probably better to pay the price at query time instead of on every refresh.

[TIP]



Make your global ordinals pay for themselves. If you have very high cardinality fields that take seconds to rebuild, increase the `refresh_interval` so that global ordinals remain valid for longer. This will also reduce CPU usage, as you will need to rebuild global ordinals less often.

=====

[[index-warmers]] ===== Index Warmers

Finally, we come to *index warmers*. Warmers((("index warmers"))) predate eager fielddata loading and eager global ordinals, but they still serve a purpose. An index warmer allows you to specify a query and aggregations that should be run before a new segment is made visible to search. The idea is to prepopulate, or *warm*, caches so your users never see a spike in latency.

Originally, the most important use for warmers was to make sure that fielddata was pre-loaded, as this is usually the most costly step. This is now better controlled with the techniques we discussed previously. However, warmers can be used to prebuild filter caches, and can still be used to preload fielddata should you so choose.

Let's register a warmer and then talk about what's happening:

[source,js]

```
PUT /music/_warmer/warmer_1 <1> { "query" : { "filtered" : { "filter" : { "bool" : { "should" : [ <2> { "term" : { "tag" : "rock" } }, { "term" : { "tag" : "hiphop" } }, { "term" : { "tag" : "electronics" } } ] } } }, "aggs" : { "price" : { "histogram" : { "field" : "price", <3> "interval" : 10 } } }
```

```
}
```

<1> Warmers are associated with an index (`music`) and are registered using the `_warmer` endpoint and a unique ID (`warmer_1`).

<2> The three most popular music genres have their filter caches prebuilt.

<3> The fielddata and global ordinals for the `price` field will be preloaded.

Warmers are registered against a specific index.((("warmers", see="index warmers"))) Each warmer is given a unique ID, because you can have multiple warmers per index.

Then you just specify a query, any query. It can include queries, filters, aggregations, sort values, scripts--literally any valid query DSL. The point is to register queries that are representative of the traffic that your users will generate, so that appropriate caches can be



prepopulated.

When a new segment is created, Elasticsearch will *literally* execute the queries registered in your warmers. The act of executing these queries will force caches to be loaded. Only after all warmers have been executed will the segment be made visible to search.

[WARNING]

Similar to eager loading, warmers shift the cost of cold caches to refresh time. When registering warmers, it is important to be judicious. You *could* add thousands of warmers to make sure every cache is populated--but that will drastically increase the time it takes for new segments to be made searchable.

In practice, select a handful of queries that represent the majority of your

user's queries and register those.

Some administrative details (such as getting existing warmers and deleting warmers) that have been omitted from this explanation. Refer to the <http://bit.ly/1AUGwys>[warmers documentation] for the rest of the details.



==== Preventing Combinatorial Explosions

The `terms` bucket dynamically builds buckets based on your data; it doesn't know up front how many buckets will be generated. (((("combinatorial explosions, preventing")))) (((("aggregations", "preventing combinatorial explosions")))) While this is fine with a single aggregation, think about what can happen when one aggregation contains another aggregation, which contains another aggregation, and so forth. The combination of unique values in each of these aggregations can lead to an explosion in the number of buckets generated.

Imagine we have a modest dataset that represents movies. Each document lists the actors in that movie:

[source,js]

```
{ "actors" : [ "Fred Jones", "Mary Jane", "Elizabeth Worthing" ] }
```

```
}
```

If we want to determine the top 10 actors and their top costars, that's trivial with an aggregation:

[source,js]

```
{ "aggs" : { "actors" : { "terms" : { "field" : "actors", "size" : 10 }, "aggs" : { "costars" : { "terms" : { "field" : "actors", "size" : 5 } } } } }
```

```
}
```

This will return a list of the top 10 actors, and for each actor, a list of their top five costars. This seems like a very modest aggregation; only 50 values will be returned!

However, this seemingly (((("aggregations", "fielddata", "datastructure overview")))) innocuous query can easily consume a vast amount of memory. You can visualize a `terms` aggregation as building a tree in memory. The `actors` aggregation will build the first level of the tree, with a bucket for every actor. Then, nested under each node in the first level, the `costars` aggregation will build a second level, with a bucket for every costar, as seen in <>. That means that a single movie will generate n^2 buckets!



[[depth-first-1]] .Build full depth tree image::images/300_120_depth_first_1.svg["Build full depth tree"]

To use some real numbers, imagine each movie has 10 actors on average. Each movie will then generate $10^2 = 100$ buckets. If you have 20,000 movies, that's roughly 2,000,000 generated buckets.

Now, remember, our aggregation is simply asking for the top 10 actors and their co-stars, totaling 50 values. To get the final results, we have to generate that tree of 2,000,000 buckets, sort it, and finally prune it such that only the top 10 actors are left. This is illustrated in <> and <>.

[[depth-first-2]] .Sort tree image::images/300_120_depth_first_2.svg["Sort tree"]

[[depth-first-3]] .Prune tree image::images/300_120_depth_first_3.svg["Prune tree"]

At this point you should be quite distraught. Twenty thousand documents is paltry, and the aggregation is pretty tame. What if you had 200 million documents, wanted the top 100 actors and their top 20 costars, as well as the costars' costars?

You can appreciate how quickly combinatorial expansion can grow, making this strategy untenable. There is not enough memory in the world to support uncontrolled combinatorial explosions.

==== Depth-First Versus Breadth-First

Elasticsearch allows you to change the *collection mode* of an aggregation, for exactly this situation. (((("collection mode")))) (((("aggregations", "preventing combinatorial explosions", "depth-first versus breadth-first")))) The strategy we outlined previously--building the tree fully and then pruning--is called *depth-first* and it is the default. (((("depth-first collection strategy")))) Depth-first works well for the majority of aggregations, but can fall apart in situations like our actors and costars example.

For these special cases, you should use an alternative collection strategy called *breadth-first*. (((("breadth-first collection strategy")))) This strategy works a little differently. It executes the first layer of aggregations, and *then* performs a pruning phase before continuing, as illustrated in <> through <>.

In our example, the `actors` aggregation would be executed first. At this point, we have a single layer in the tree, but we already know who the top 10 actors are! There is no need to keep the other actors since they won't be in the top 10 anyway.

[[breadth-first-1]] .Build first level image::images/300_120_breadth_first_1.svg["Build first level"]



[[breadth-first-2]] .Sort first level image::images/300_120_breadth_first_2.svg["Sort first level"]

[[breadth-first-3]] .Prune first level image::images/300_120_breadth_first_3.svg["Prune first level"]

Since we already know the top ten actors, we can safely prune away the rest of the long tail. After pruning, the next layer is populated based on *its* execution mode, and the process repeats until the aggregation is done, as illustrated in <>. This prevents the combinatorial explosion of buckets and drastically reduces memory requirements for classes of queries that are amenable to breadth-first.

[[breadth-first-4]] .Populate full depth for remaining nodes

image::images/300_120_breadth_first_4.svg["Step 4: populate full depth for remaining nodes"]

To use breadth-first, simply (((("collect parameter, enabling breadth-first"))))enable it via the `collect` parameter:

[source,js]

```
{ "aggs" : { "actors" : { "terms" : { "field" : "actors", "size" : 10, "collect_mode" : "breadth_first"  
<1> }, "aggs" : { "costars" : { "terms" : { "field" : "actors", "size" : 5 } } } } }
```

```
}
```

<1> Enable `breadth_first` on a per-aggregation basis.

Breadth-first should be used only when you expect more buckets to be generated than documents landing in the buckets. Breadth-first works by caching document data at the bucket level, and then replaying those documents to child aggregations after the pruning phase.

The memory requirement of a breadth-first aggregation is linear to the number of documents in each bucket prior to pruning. For many aggregations, the number of documents in each bucket is very large. Think of a histogram with monthly intervals: you might have thousands or hundreds of thousands of documents per bucket. This makes breadth-first a bad choice, and is why depth-first is the default.

But for the actor example--which generates a large number of buckets, but each bucket has relatively few documents--breadth-first is much more memory efficient, and allows you to build aggregations that would otherwise fail.



== Closing Thoughts

This section covered a lot of ground, and a lot of deeply technical issues. Aggregations bring a power and flexibility to Elasticsearch that is hard to overstate. The ability to nest buckets and metrics, to quickly approximate cardinality and percentiles, to find statistical anomalies in your data, all while operating on near-real-time data and in parallel to full-text search--these are game-changers to many organizations.

It is a feature that, once you start using it, you'll find dozens of other candidate uses. Real-time reporting and analytics is central to many organizations (be it over business intelligence or server logs).

But with great power comes great responsibility, and for Elasticsearch that often means proper memory stewardship. Memory is often the limiting factor in Elasticsearch deployments, particularly those that heavily utilize aggregations.

Because aggregation data is loaded to `fielddata`--and this is an in-memory data structure--managing ((`"aggregations"`, `"managing efficient memory usage"`))efficient memory usage is important.

The management of this memory can take several forms, depending on your particular use-case:

- At a data level, by making sure you `analyze` (or `not_analyze`) your data appropriately so that it is memory-friendly
- During indexing, by configuring heavy fields to use disk-based doc values instead of in-memory `fielddata`
- At search time, by utilizing approximate aggregations and data filtering
- At a node level, by setting hard memory and dynamic circuit-breaker limits
- At an operations level, by monitoring memory usage and controlling slow garbage-collection cycles, potentially by adding more nodes to the cluster

Most deployments will use one or more of the preceding methods. The exact combination is highly dependent on your particular environment. Some organizations need blisteringly fast responses and opt to simply add more nodes. Other organizations are limited by budget and choose doc values and approximate aggregations.

Whatever the path you take, it is important to assess the available options and create both a short- and long-term plan. Decide how your memory situation exists today and what (if anything) needs to be done. Then decide what will happen in six months or one year as your data grows. What methods will you use to continue scaling?

It is better to plan out these life cycles of your cluster ahead of time, rather than panicking at 3 a.m. because your cluster is at 90% heap utilization.



== High-level concepts

Like the query DSL, aggregations have a *composable* syntax: independent units of functionality can be mixed and matched to provide the custom behavior that you need. This means that there are only a few basic concepts to learn, but nearly limitless combinations of those basic components.

To master aggregations, you only need to understand two main concepts:

Buckets:: Collections of documents which meet a criterion. *Metrics*:: Statistics calculated on the documents in a bucket.

That's it! Every aggregation is simply a combination of one or more buckets and zero or more metrics. To translate into rough SQL terms:

[source,sql]

`SELECT COUNT(color) <1> FROM table`

GROUP BY color <2>

<1> `COUNT(color)` is equivalent to a metric

<2> `GROUP BY color` is equivalent to a bucket

Buckets are conceptually similar to grouping in SQL, while metrics are similar to `COUNT()`, `SUM()`, `MAX()`, etc.

Let's dig into both of these concepts and see what they entail.

==== Buckets

A bucket is simply a collection of documents that meet a certain criteria.

- An employee would land in either the "male" or "female" bucket.
- The city of Albany would land in the "New York" state bucket.
- The date "2014-10-28" would land within the "October" bucket.

As aggregations are executed, the values inside each document are evaluated to determine if they match a bucket's criteria. If they match, the document is placed inside the bucket and the aggregation continues.

Buckets can also be nested inside of other buckets, giving you a hierarchy or conditional partitioning scheme. For example, "Cincinnati" would be placed inside the "Ohio" state bucket, and the *entire* "Ohio" bucket would be placed inside the "USA" country bucket.



There are a variety of different buckets in Elasticsearch, which allow you to partition documents in many different ways (by hour, by most popular terms, by age ranges, by geographical location, etc.). But fundamentally they all operate on the same principle: partitioning documents based on a criteria.

==== Metrics

Buckets allow us to partition documents into useful subsets, but ultimately what we want is some kind of *metric* calculated on those documents in each bucket.

Bucketing is the means to an end - it provides a way to group documents in a way that you can calculate interesting metrics.

Most metrics are simple mathematical operations (min, mean, max, sum, etc.) which are calculated using the document values. In practical terms, metrics allow you to calculate quantities such as the average salary, or the maximum sale price, or the 95th percentile for query latency.

==== Combining the two

An aggregation is a combination of buckets and metrics. An aggregation may have a single bucket, or a single metric, or one of each. It may even have multiple buckets nested inside of other buckets.

For example, we can partition documents by which country they belong to (a bucket), then calculate the average salary per country (a metric).

Because buckets can be nested, we can derive a much more complex aggregation:

1. Partition documents by country (bucket)
2. Then partition each country bucket by gender (bucket)
3. Then partition each gender bucket by age ranges (bucket)
4. Finally, calculate the average salary for each age range (metric)

This will give you the average salary per combination. All in one request and with one pass over the data!



== Aggregation Test-Drive

We could spend the next few pages defining the various aggregations and their syntax, (((“aggregations”, “basic example”, id=“ix_basicex”))) but aggregations are truly best learned by example. Once you learn how to think about aggregations, and how to nest them appropriately, the syntax is fairly trivial.

[NOTE]

A complete list of aggregation buckets and metrics can be found at the [http://bit.ly/1KNL1R3\[online reference documentation\]](http://bit.ly/1KNL1R3). We'll cover many of them in this chapter, but glance

over it after finishing so you are familiar with the full range of capabilities.

So let's just dive in and start with an example. We are going to build some aggregations that might be useful to a car dealer. Our data will be about car transactions: the car model, manufacturer, sale price, when it sold, and more.

First we will bulk-index some data to work with:

[source,js]

```
POST /cars/transactions/_bulk { "index": {} } { "price" : 10000, "color" : "red", "make" : "honda", "sold" : "2014-10-28" } { "index": {} } { "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05" } { "index": {} } { "price" : 30000, "color" : "green", "make" : "ford", "sold" : "2014-05-18" } { "index": {} } { "price" : 15000, "color" : "blue", "make" : "toyota", "sold" : "2014-07-02" } { "index": {} } { "price" : 12000, "color" : "green", "make" : "toyota", "sold" : "2014-08-19" } { "index": {} } { "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05" } { "index": {} } { "price" : 80000, "color" : "red", "make" : "bmw", "sold" : "2014-01-01" } { "index": {} }
```

{ “price” : 25000, “color” : “blue”, “make” : “ford”, “sold” : “2014-02-12” }

```
// SENSE: 300_Aggregations/20_basic_example.json
```



Now that we have some data, let's construct our first aggregation. A car dealer may want to know which color car sells the best. This is easily accomplished using a simple aggregation. We will do this using a `terms` bucket:

[source.js]

```
GET /cars/transactions/_search?search_type=count { "aggs" : { <1> "colors" : { <2> "terms" : { "field" : "color" <3> } } }}
```

```
}
```

```
// SENSE: 300_Aggregations/20_basic_example.json
```

<1> Aggregations are placed under the ((("aggregations", "aggs parameter")))top-level `aggs` parameter (the longer `aggregations` will also work if you prefer that).

<2> We then name the aggregation whatever we want: `colors`, in this example

<3> Finally, we define a single bucket of type `terms`.

Aggregations are executed in the context of search results,((("searching", "aggregations executed in context of search results"))) which means it is just another top-level parameter in a search request (for example, using the `/_search` endpoint). Aggregations can be paired with queries, but we'll tackle that later in <<`_scoping_aggregations`>>.

[NOTE]

You'll notice that we used the `count <>.((("count search type")))` Because we don't care about search results--the aggregation totals--the

count `search_type` will be faster because it omits the fetch phase.

Next we define a name for our aggregation. Naming is up to you; the response will be labeled with the name you provide so that your application can parse the results later.



Next we define the aggregation itself. For this example, we are defining a single `terms` bucket.`((("buckets", "terms bucket (example)"))((("terms bucket", "defining in example aggregation"))))` The `terms` bucket will dynamically create a new bucket for every unique term it encounters. Since we are telling it to use the `color` field, the `terms` bucket will dynamically create a new bucket for each color.

Let's execute that aggregation and take a look at the results:

[source.js]

```
{ ... "hits": { "hits": [] <1> }, "aggregations": { "colors": { <2> "buckets": [ { "key": "red", <3> "doc_count": 4 <4> }, { "key": "blue", "doc_count": 2 }, { "key": "green", "doc_count": 2 } ] } }
```

```
}
```

<1> No search hits are returned because we used the `search_type=count` parameter

<2> Our `colors` aggregation is returned as part of the `aggregations` field.

<3> The `key` to each bucket corresponds to a unique term found in the `color` field. It also always includes `doc_count`, which tells us the number of docs containing the term.

<4> The count of each bucket represents the number of documents with this color.

The `((("doc_count")))` response contains a list of buckets, each corresponding to a unique color (for example, red or green). Each bucket also includes a count of the number of documents that "fell into" that particular bucket. For example, there are four red cars.

The preceding example is operating entirely in real time: if the documents are searchable, they can be aggregated. This means you can take the aggregation results and pipe them straight into a graphing library to generate real-time dashboards. As soon as you sell a silver car, your graphs would dynamically update to include statistics about silver cars.

Voila! Your first aggregation! `((("aggregations", "basic example", startref ="ix_basicex")))`



==== Adding a Metric to the Mix

The previous example told us the number of documents in each bucket, which is useful. (((("aggregations", "basic example", "adding a metric")))) But often, our applications require more-sophisticated metrics about the documents. (((("metrics", "adding to basic aggregation example")))) For example, what is the average price of cars in each bucket?

To get this information, we need to tell Elasticsearch which metrics to calculate, and on which fields. (((("buckets", "nesting metrics in")))) This requires *nesting* metrics inside the buckets. Metrics will calculate mathematical statistics based on the values of documents within a bucket.

Let's go ahead and add (((("average metric")))) an `average` metric to our car example:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs": { "colors": { "terms": { "field": "color" }, "aggs": { <1> "avg_price": { <2> "avg": { "field": "price" <3> } } } } }
```

```
}
```

// SENSE: 300_Aggregations/20_basic_example.json

<1> We add a new `aggs` level to hold the metric.

<2> We then give the metric a name: `avg_price`.

<3> And finally, we define it as an `avg` metric over the `price` field.

As you can see, we took the previous example and tacked on a new `aggs` level. This new aggregation level allows us to nest the `avg` metric inside the `terms` bucket. Effectively, this means we will generate an average for each color.

Just like the `colors` example, we need to name our metric (`avg_price`) so we can retrieve the values later. Finally, we specify the metric itself (`avg`) and what field we want the average to be calculated on (`price`):

[source,js]

```
{ ... "aggregations": { "colors": { "buckets": [ { "key": "red", "doc_count": 4, "avg_price": { <1> "value": 32500 } }, { "key": "blue", "doc_count": 2, "avg_price": { "value": 20000 } }, { "key": "green", "doc_count": 2, "avg_price": { "value": 21000 } } ] } } ...
```



<1> New `avg_price` element in response

Although the response has changed minimally, the data we get out of it has grown substantially. Before, we knew there were four red cars. Now we know that the average price of red cars is \$32,500. This is something that you can plug directly into reports or graphs.



==== Buckets Inside Buckets

The true power of aggregations becomes apparent once you start playing with different nesting schemes.((("aggregations", "basic example", "buckets nested in other buckets"))) ((("buckets", "nested in other buckets"))) In the previous examples, we saw how you could nest a metric inside a bucket, which is already quite powerful.

But the real exciting analytics come from nesting buckets inside *other buckets*. This time, we want to find out the distribution of car manufacturers for each color:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs": { "colors": { "terms": { "field": "color" }, "aggs": { "avg_price": { <1> "avg": { "field": "price" } }, "make": { <2> "terms": { "field": "make" <3> } } } } }
```

```
}
```

// SENSE: 300_Aggregations/20_basic_example.json

<1> Notice that we can leave the previous `avg_price` metric in place.

<2> Another aggregation named `make` is added to the `color` bucket.

<3> This aggregation is a `terms` bucket and will generate unique buckets for each car make.

A few interesting things happened here.((("metrics", "independent, on levels of an aggregation"))) First, you'll notice that the previous `avg_price` metric is left entirely intact. Each *level* of an aggregation can have many metrics or buckets. The `avg_price` metric tells us the average price for each car color. This is independent of other buckets and metrics that are also being built.

This is important for your application, since there are often many related, but entirely distinct, metrics that you need to collect. Aggregations allow you to collect all of them in a single pass over the data.

The other important thing to note is that the aggregation we added, `make`, is a `terms` bucket (nested inside the `colors` `terms` bucket). This means we will((("terms bucket", "nested in another terms bucket"))) generate a (`color`, `make`) tuple for every unique combination in your dataset.

Let's take a look at the response (truncated for brevity, since it is now growing quite long):



[source,js]

```
{ ... "aggregations": { "colors": { "buckets": [ { "key": "red", "doc_count": 4, "make": { <1> "buckets": [ { "key": "honda", <2> "doc_count": 3 }, { "key": "bmw", "doc_count": 1 } ] }, "avg_price": { "value": 32500 <3> } } }, ... }
```

<1> Our new aggregation is nested under each color bucket, as expected.

<2> We now see a breakdown of car makes for each color.

<3> Finally, you can see that our previous `avg_price` metric is still intact.

The response tells us the following:

- There are four red cars.
- The average price of a red car is \$32,500.
- Three of the red cars are made by Honda, and one is a BMW.



==== One Final Modification

Just to drive the point home, let's make one final modification to our example before moving on to new topics.((("aggregations", "basic example", "adding extra metrics")))((("metrics", "adding more to aggregation (example)")))) Let's add two metrics to calculate the min and max price for each make:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs": { "colors": { "terms": { "field": "color" }, "aggs": { "avg_price": { "avg": { "field": "price" } }, "make" : { "terms" : { "field" : "make" }, "aggs" : { <1> "min_price" : { "min": { "field": "price" } }, <2> "max_price" : { "max": { "field": "price" } } <3> } } } } }
```

}

// SENSE: 300_Aggregations/20_basic_example.json

<1> We need to add another `aggs` level for nesting.

<2> Then we include a `min` metric.

<3> And a `max` metric.

Which gives ((("min and max metrics (aggregation example)"))))us the following output (again, truncated):

[source,js]

```
{ ... "aggregations": { "colors": { "buckets": [ { "key": "red", "doc_count": 4, "make": { "buckets": [ { "key": "honda", "doc_count": 3, "min_price": { "value": 10000 <1> }, "max_price": { "value": 20000 <1> } }, { "key": "bmw", "doc_count": 1, "min_price": { "value": 80000 }, "max_price": { "value": 80000 } } ] }, "avg_price": { "value": 32500 } },
```

...

<1> The `min` and `max` metrics that we added now appear under each `make`

With those two buckets, we've expanded the information derived from this query to include the following:



- There are four red cars.
- The average price of a red car is \$32,500.
- Three of the red cars are made by Honda, and one is a BMW.
- The cheapest red Honda is \$10,000.
- The most expensive red Honda is \$20,000.



// I'd limit this list to the metrics and rely on the obvious. You don't need to explain what min/max/avg etc are. Then say that we'll discuss these more interesting metrics in later chapters: cardinality, percentiles, significant terms. The buckets I'd mention under the relevant section, eg Histo & Range, etc

== Available Buckets and Metrics

There are a number of different buckets and metrics. The reference documentation does a great job describing the various parameters and how they affect the component. Instead of re-describing them here, we are simply going to link to the reference docs and provide a brief description. Skim the list so that you know what is available, and check the reference docs when you need exact parameters.

[float] === Buckets

- [{ref}search-aggregations-bucket-global-aggregation.html\[Global\]](#): includes all documents
- [{ref}search-aggregations-bucket-filter-aggregation.html\[Filter\]](#): only includes document the filter
- [{ref}search-aggregations-bucket-missing-aggregation.html\[Missing\]](#): all documents which a particular field
- [{ref}search-aggregations-bucket-terms-aggregation.html\[Terms\]](#): generates a new bucket f
- [{ref}search-aggregations-bucket-range-aggregation.html\[Range\]](#): creates arbitrary ranges fall into
- [{ref}search-aggregations-bucket-daterange-aggregation.html\[Date Range\]](#): similar to Rang aware
- [{ref}search-aggregations-bucket-iprange-aggregation.html\[IPV4 Range\]](#): similar to Range,
- [{ref}search-aggregations-bucket-geodistance-aggregation.html\[Geo Distance\]](#): similar to geo points
- [{ref}search-aggregations-bucket-histogram-aggregation.html\[Histogram\]](#): equal-width, dyn
- [{ref}search-aggregations-bucket-datehistogram-aggregation.html\[Date Histogram\]](#): similar calendar aware
- [{ref}search-aggregations-bucket-nested-aggregation.html\[Nested\]](#): a special bucket for w nested documents (see <>nested-aggregation>>)
- [{ref}search-aggregations-bucket-geohashgrid-aggregation.html\[Geohash Grid\]](#): partitions what geohash grid they fall into (see <>geohash-grid-agg>>)
- [{ref}search-aggregations-metrics-top-hits-aggregation.html\[TopHits\]](#): Return the top sea

[float] === Metrics



- Individual statistics: [{ref}search-aggregations-metrics-min-aggregation.html\[Min\]](#), [{ref}search-aggregations-metrics-stats-aggregation.html\[Stats\]](#): calculates min/mean/max
- [{ref}search-aggregations-metrics-extendedstats-aggregation.html\[Extended Stats\]](#): Same as above
- [{ref}search-aggregations-metrics-valuecount-aggregation.html\[Value Count\]](#): calculates the number of distinct values in a bucket, which can be different from the number of documents (e.g. multi-valued fields)
- [{ref}search-aggregations-metrics-cardinality-aggregation.html\[Cardinality\]](#): calculates the number of documents in a bucket
- [{ref}search-aggregations-metrics-percentile-aggregation.html\[Percentiles\]](#): calculates the specified percentile of numeric values in a bucket (see <>percentiles>>)
- [{ref}search-aggregations-bucket-significantterms-aggregation.html\[Significant Terms\]](#): finds the most significant terms in a bucket (see <>significant-terms>>)





== Building Bar Charts

One of the exciting aspects of aggregations are how easily they are converted into charts and graphs.((("bar charts", building from aggregations", id="ix_barcharts", range="startofrange")))((("aggregations", "building bar charts from"))) In this chapter, we are focusing on various analytics that we can wring out of our example dataset. We will also demonstrate the types of charts aggregations can power.

The `++histogram++` bucket is particularly useful.((("buckets", "histogram")))((("histogram bucket"))))((("histograms"))) Histograms are essentially bar charts, and if you've ever built a report or analytics dashboard, you undoubtedly had a few bar charts in it. The histogram works by specifying an interval. If we were histogramming sale prices, you might specify an interval of 20,000. This would create a new bucket every \$20,000. Documents are then sorted into buckets.

For our dashboard, we want to know how many cars sold in each price range. We would also like to know the total revenue generated by that price bracket. This is calculated by summing the price of each car sold in that interval.

To do this, we use a `histogram` and a nested `sum` metric:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs":{ "price":{ "histogram":{ <1>  
"field": "price", "interval": 20000 }, "aggs":{ "revenue": { "sum": { <2> "field" : "price" } } } } }
```

```
}
```

```
// SENSE: 300_Aggregations/30_histogram.json
```

<1> The `histogram` bucket requires two parameters: a numeric field, and an interval that defines the bucket size. // Mention use of "size" to get back just the top result?

<2> A `sum` metric is nested inside each price range, which will show us the total revenue for that bracket

As you can see, our query is built around the `price` aggregation, which contains a `histogram` bucket. This bucket requires a numeric field to calculate buckets on, and an interval size. The interval defines how "wide" each bucket is. An interval of 20000 means we will have the ranges `[0-19999, 20000-39999, ...]`.



Next, we define a nested metric inside the histogram. This is a `sum` metric, which will sum up the `price` field from each document landing in that price range. This gives us the revenue for each price range, so we can see if our business makes more money from commodity or luxury cars.

And here is the response:

[source,js]

```
{ ... "aggregations": { "price": { "buckets": [ { "key": 0, "doc_count": 3, "revenue": { "value": 37000 } }, { "key": 20000, "doc_count": 4, "revenue": { "value": 95000 } }, { "key": 80000, "doc_count": 1, "revenue": { "value": 80000 } } ] } }
```

```
}
```

The response is fairly self-explanatory, but it should be noted that the histogram keys correspond to the lower boundary of the interval. The key `0` means `0-19,999`, the key `20000` means `20,000-39,999`, and so forth.

[NOTE]

You'll notice that empty intervals, such as \$40,000-60,000, is missing in the response. The `histogram` bucket omits these by default, since it could lead to the unintended generation of potentially enormous output.

**We'll discuss how to include empty buckets in the next section,
<<_returning_empty_buckets>>.**

Graphically, you could represent the preceding data in the histogram shown in <>.

```
[[barcharts-histo1]] .Sales and Revenue per price bracket  
image::images/elas_28in01.png["Sales and Revenue per price bracket"]
```

Of course, you can build bar charts with any aggregation that emits categories and statistics, not just the `histogram` bucket. Let's build a bar chart of popular makes, and their average price, and then calculate the standard error to add error bars on our chart. This will use the



```
terms bucket and an extended_stats (((\"extended_stats metric\")))metric:
```

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs": { "makes": { "terms": { "field": "make", "size": 10 }, "aggs": { "stats": { "extended_stats": { "field": "price" } } } } }
```

```
}
```

This will return a list of makes (sorted by popularity) and a variety of statistics about each. In particular, we are interested in `stats.avg`, `stats.count`, and `stats.std_deviation`.

Using(((\"standard error, calculating\"))) this information, we can calculate the standard error:

```
..... std_err = std_deviation / count .....
```

This will allow us to build a chart like <>.

[[barcharts-bar1]] .Average price of all makes, with error bars

image::images/elas_28in02.png["Average price of all makes, with error bars"]

```
((("bar charts, building from aggregations", range="endofrange", startref="ix_barcharts")))
```



== Looking at Time

If search is the most popular activity in Elasticsearch, building date histograms must be the second most popular.((("date histograms", "building")))(("histograms", "building date histograms"))((("aggregations", "building date histograms from"))) Why would you want to use a date histogram?

Imagine your data has a timestamp.(("time", "analytics over", id="ixtimeanalyze")) It doesn't matter what the data is--Apache log events, stock buy/sell transaction dates, baseball game times--anything with a timestamp can benefit from the date histogram. When you have a timestamp, you often want to build metrics that are expressed _over time:

- How many cars sold each month this year?
- What was the price of this stock for the last 12 hours?
- What was the average latency of our website every hour in the last week?

While regular histograms are often represented as bar charts, date histograms tend to be converted into line graphs representing time series.((("analytics", "over time"))) Many companies use Elasticsearch solely for analytics over time series data. The `date_histogram` bucket is their bread and butter.

The `date_histogram` bucket works((("buckets", "date_histogram"))) similarly to the regular `histogram`. Rather than building buckets based on a numeric field representing numeric ranges, it builds buckets based on time ranges. Each bucket is therefore defined as a certain calendar size (for example, `1 month` or `2.5 days`).

[role="pagebreak-before"] .Can a Regular Histogram Work with Dates?

Technically, yes.((("histogram bucket", "dates and"))) A regular `histogram` bucket will work with dates. However, it is not calendar-aware. With the `date_histogram`, you can specify intervals such as `1 month`, which knows that February is shorter than December. The `date_histogram` also has the advantage of being able to work with time zones, which allows you to customize graphs to the time zone of the user, not the server.

The regular `histogram` will interpret dates as numbers, which means you must specify intervals in terms of milliseconds. And the aggregation doesn't know about calendar intervals, which makes it largely useless for dates.

Our first example ((("line charts", "building from aggregations")))will build a simple line chart to answer this question: how many cars were sold each month?



[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs": { "sales": { "date_histogram": {  
"field": "sold", "interval": "month", <1> "format": "yyyy-MM-dd" <2> } } } }
```

```
}
```

// SENSE: 300_Aggregations/35_date_histogram.json

<1> The interval is requested in calendar terminology (for example, one month per bucket).

// "pretty"-> "readable by humans". mention that otherwise get back ms-since-epoch?

<2> We provide a date format so that bucket keys are pretty.

Our query has a single aggregation, which builds a bucket per month. This will give us the number of cars sold in each month. An additional `format` parameter is provided so the buckets have "pretty" keys. Internally, dates are simply represented as a numeric value. This tends to make UI designers grumpy, however, so a prettier format can be specified using common date formatting.

The response is both expected and a little surprising (see if you can spot the surprise):

[source,js]

```
{ ... "aggregations": { "sales": { "buckets": [ { "key_as_string": "2014-01-01", "key":  
1388534400000, "doc_count": 1 }, { "key_as_string": "2014-02-01", "key": 1391212800000,  
"doc_count": 1 }, { "key_as_string": "2014-05-01", "key": 1398902400000, "doc_count": 1 }, {  
"key_as_string": "2014-07-01", "key": 1404172800000, "doc_count": 1 }, { "key_as_string":  
"2014-08-01", "key": 1406851200000, "doc_count": 1 }, { "key_as_string": "2014-10-01",  
"key": 1412121600000, "doc_count": 1 }, { "key_as_string": "2014-11-01", "key":  
1414800000000, "doc_count": 2 } ] ... }
```

```
}
```

The aggregation is represented in full. As you can see, we have buckets that represent months, a count of docs in each month, and our pretty `key_as_string`.

`[[_returning_empty_buckets]]` === Returning Empty Buckets

Notice something odd about that last response?



Yep, that's right.((("aggregations", "returning empty buckets")))((("buckets", "empty", "returning"))) We are missing a few months! By default, the `date_histogram` (and `histogram` too) returns only buckets that have a nonzero document count.

This means your histogram will be a minimal response. Often, this is not the behavior you want. For many applications, you would like to dump the response directly into a graphing library without doing any post-processing.

Essentially, we want buckets even if they have a count of zero. We can set two additional parameters that will provide this behavior:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs": { "sales": { "date_histogram": {  
"field": "sold", "interval": "month", "format": "yyyy-MM-dd", "min_doc_count" : 0, <1>  
"extended_bounds" : { <2> "min" : "2014-01-01", "max" : "2014-12-31" } } } }  
}  
  
// SENSE: 300_Aggregations/35_date_histogram.json
```

<1> This parameter forces empty buckets to be returned.

<2> This parameter forces the entire year to be returned.

The two additional parameters will force the response to return all months in the year, regardless of their doc count.((("min_doc_count parameter"))) The `min_doc_count` is very understandable: it forces buckets to be returned even if they are empty.

The `extended_bounds` parameter requires a little explanation.((("extended_bounds parameter"))) The `min_doc_count` parameter forces empty buckets to be returned, but by default Elasticsearch will return only buckets that are between the minimum and maximum value in your data.

So if your data falls between April and July, you'll have buckets representing only those months (empty or otherwise). To get the full year, we need to tell Elasticsearch that we want buckets even if they fall *before* the minimum value or *after* the maximum value.

The `extended_bounds` parameter does just that. Once you add those two settings, you'll get a response that is easy to plug straight into your graphing libraries and give you a graph like <>.

```
[[date-histo-ts1]] .Cars sold over time image::images/elas_29in01.png["Cars sold over time"]
```



==== Extended Example

Just as we've seen a dozen times already, buckets can be nested in buckets for more-sophisticated behavior.((("buckets", "nested in other buckets", "extended example"))) ((("aggregations", "extended example"))) For illustration, we'll build an aggregation that shows the total sum of prices for all makes, listed by quarter. Let's also calculate the sum of prices per individual make per quarter, so we can see which car type is bringing in the most money to our business:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs": { "sales": { "date_histogram": {  
"field": "sold", "interval": "quarter", <1> "format": "yyyy-MM-dd", "min_doc_count": 0,  
"extended_bounds": { "min": "2014-01-01", "max": "2014-12-31" } }, "aggs": {  
"per_make_sum": { "terms": { "field": "make" }, "aggs": { "sum_price": { "sum": { "field": "price" } <2> } } }, "total_sum": { "sum": { "field": "price" } <3> } } } }
```

}

// SENSE: 300_Aggregations/35_date_histogram.json

<1> Note that we changed the interval from month to quarter .

<2> Calculate the sum per make.

<3> And the total sum of all makes combined together.

This returns a (heavily truncated) response:

[source,js]

```
{ .... "aggregations": { "sales": { "buckets": [ { "key_as_string": "2014-01-01", "key": 1388534400000, "doc_count": 2, "total_sum": { "value": 105000 }, "per_make_sum": { "buckets": [ { "key": "bmw", "doc_count": 1, "sum_price": { "value": 80000 } }, { "key": "ford", "doc_count": 1, "sum_price": { "value": 25000 } ] } } } }, ... }
```

}



We can take this response and put it into a graph, (((("line charts, building from aggregations"))))((("bar charts, building from aggregations")))) showing a line chart for total sale price, and a bar chart for each individual make (per quarter), as shown in <>.

[[date-histo-ts2]] .Sales per quarter, with distribution per make

image::images/elas_29in02.png["Sales per quarter, with distribution per make"]

==== The Sky's the Limit

These were obviously simple examples, but the sky really is the limit when it comes to charting aggregations. (((("dashboards", "building from aggregations"))))((("Kibana", "dashboard in")))) For example, <> shows a dashboard in Kibana built with a variety of aggregations.

[[kibana-img]] .Kibana--a real time analytics dashboard built with aggregations

image::images/elas_29in03.png["Kibana - a real time analytics dashboard built with aggregations"]

Because of the real-time nature of aggregations, dashboards like this are easy to query, manipulate, and interact with. This makes them ideal for nontechnical employees and analysts who need to analyze the data but cannot build a Hadoop job.

To build powerful dashboards like Kibana, however, you'll likely need some of the more advanced concepts such as scoping, filtering, and sorting aggregations. (((("time, analytics over", startref ="ix_timeanalyze"))))



[_scoping_aggregations]] == Scoping Aggregations

With all of the aggregation examples given so far, you may have noticed that we omitted a `query` from the search request. (((("queries", "in aggregations"))))(((("aggregations", "scoping")))) The entire request was simply an aggregation.

Aggregations can be run at the same time as search requests, but you need to understand a new concept: `scope`. (((("scoping aggregations", id="ix_scopeaggs", range="startofrange")))) By default, aggregations operate in the same scope as the query. Put another way, aggregations are calculated on the set of documents that match your query.

Let's look at one of our first aggregation examples:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs" : { "colors" : { "terms" : { "field" : "color" } } }
```

```
}
```

```
// SENSE: 300_Aggregations/40_scope.json
```

You can see that the aggregation is in isolation. In reality, Elasticsearch assumes "no query specified" is equivalent to "query all documents." The preceding query is internally translated as follows:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "query" : { "match_all" : {} }, "aggs" : { "colors" : { "terms" : { "field" : "color" } } }
```

```
}
```

```
// SENSE: 300_Aggregations/40_scope.json
```

The aggregation always operates in the scope of the query, so an isolated aggregation really operates in the scope of (((("match_all query", "isolated aggregations in scope of")))) a `match_all` query--that is to say, all documents.



Once armed with the knowledge of scoping, we can start to customize aggregations even further. All of our previous examples calculated statistics about *all* of the data: top-selling cars, average price of all cars, most sales per month, and so forth.

With scope, we can ask questions such as "How many colors are Ford cars available in?" We do this by simply adding a query to the request (in this case a `match` query):

[source,js]

```
GET /cars/transactions/_search <1> { "query" : { "match" : { "make" : "ford" } }, "aggs" : { "colors" : { "terms" : { "field" : "color" } } }
```

```
}
```

```
// SENSE: 300_Aggregations/40_scope.json
```

<1> We are omitting `search_type=count` so(((`"search_type", "count"`))) that search hits are returned too.

By omitting the `search_type=count` this time, we can see both the search results and the aggregation results:

[source,js]

```
{ ... "hits": { "total": 2, "max_score": 1.6931472, "hits": [ { "_source": { "price": 25000, "color": "blue", "make": "ford", "sold": "2014-02-12" } }, { "_source": { "price": 30000, "color": "green", "make": "ford", "sold": "2014-05-18" } } ], "aggregations": { "colors": { "buckets": [ { "key": "blue", "doc_count": 1 }, { "key": "green", "doc_count": 1 } ] } }}
```

```
}
```

This may seem trivial, but it is the key to advanced and powerful dashboards. You can transform any static dashboard into a real-time data exploration device by adding a search bar.((("dashboards", "adding a search bar"))) This allows the user to search for terms and see all of the graphs (which are powered by aggregations, and thus scoped to the query) update in real time. Try that with Hadoop!

[float] === Global Bucket



You'll often want your aggregation to be scoped to your query. But sometimes you'll want to search for a subset of data, but aggregate across *all* of your data.((("aggregations", "scoping", "global bucket")))((("scoping aggregations", "using a global bucket"))))

For example, say you want to know the average price of Ford cars compared to the average price of *all* cars. We can use a regular aggregation (scoped to the query) to get the first piece of information. The second piece of information can be obtained by using((("buckets", "global")))((("global bucket")))) a `global` bucket.

The `+global+` bucket will contain *all* of your documents, regardless of the query scope; it bypasses the scope completely. Because it is a bucket, you can nest aggregations inside it as usual:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "query" : { "match" : { "make" : "ford" } }, "aggs" : { "single_avg_price": { "avg" : { "field" : "price" } <1> }, "all": { "global" : {}, <2> "aggs" : { "avg_price": { "avg" : { "field" : "price" } <3> } }
```

```
        }
    }
}
```

```
}
```

```
// SENSE: 300_Aggregations/40_scope.json
```

<1> This aggregation operates in the query scope (for example, all docs matching `+ford+`)

<2> The `global` bucket has no parameters.

<3> This aggregation operates on the all documents, regardless of the make.

The `+single_avg_price+` metric calculation is based on all documents that fall under the query scope--all `+ford+` cars. The `+avg_price+` metric is nested under a `global` bucket, which means it ignores scoping entirely and calculates on all the documents. The average returned for that aggregation represents the average price of all cars.

If you've made it this far in the book, you'll recognize the mantra: use a filter wherever you can. The same applies to aggregations, and in the next chapter we show you how to filter an aggregation instead of just limiting the query scope.((("scoping aggregations", "range="endofrange", startref="ix_scopeaggs")))



== Filtering Queries and Aggregations

A natural extension to aggregation scoping is filtering. Because the aggregation operates in the context of the query scope, any filter applied to the query will also apply to the aggregation.

[float="true"] === Filtered Query If we want to find all cars over \$10,000 and also calculate the average price for those cars,(((("filtering", "serch query results"))))((("filtered query")))((("queries", "filtered")))) we can simply use a `filtered` query:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "query" : { "filtered": { "filter": { "range": { "price": { "gte": 10000 } } } } }, "aggs" : { "single_avg_price": { "avg" : { "field" : "price" } } }
```

```
}
```

```
// SENSE: 300_Aggregations/45_filtering.json
```

Fundamentally, using a `filtered` query is no different from using a `match` query, as we discussed in the previous chapter. The query (which happens to include a filter) returns a certain subset of documents, and the aggregation operates on those documents.

[float="true"] === Filter Bucket

But what if you would like to filter just the aggregation results?(((("filtering", "aggregation results, not the query"))))((("aggregations", "filtering just aggregations")))) Imagine we are building the search page for our car dealership. We want to display search results according to what the user searches for. But we also want to enrich the page by including the average price of cars (matching the search) that were sold in the last month.

We can't use simple scoping here, since there are two different criteria. The search results must match `+ford+`, but the aggregation results must match `+ford+ AND +sold > now - 1M+`.

To solve this problem, we can use a special bucket called `filter .((("filter bucket"))))((("buckets", "filter"))))` You specify a filter, and when documents match the filter's criteria, they are added to the bucket.

Here is the resulting query:

[source,js]



```
GET /cars/transactions/_search?search_type=count { "query":{ "match": { "make": "ford" } },  
"aggs":{ "recent_sales": { "filter": { <1> "range": { "sold": { "from": "now-1M" } } } }, "aggs": {  
"average_price":{ "avg": { "field": "price" <2> } } } } }
```

// SENSE: 300_Aggregations/45_filtering.json

<1> Using the `filter` bucket to apply a filter in addition to the `query` scope.

<2> This `avg` metric will therefore average only docs that are both `+ford+` and sold in the last month.

Since the `filter` bucket operates like any other bucket, you are free to nest other buckets and metrics inside. All nested components will "inherit" the filter. This allows you to filter selective portions of the aggregation as required.

[float="true"] === Post Filter

So far, we have a way to filter both the search results and aggregations (a `filtered` query), as well as filtering individual portions of the aggregation (`filter` bucket).

You may be thinking to yourself, "hmm...is there a way to filter *just* the search results but not the aggregation?"(("filtering", "search results, not the aggregation"))((("post filter"))) The answer is to use a `post_filter`.

This is a top-level search-request element that accepts a filter. The filter is applied *after* the query has executed (hence the `+post+` moniker: it runs *post query execution*). Because it operates after the query has executed, it does not affect the query scope--and thus does not affect the aggregations either.

We can use this behavior to apply additional filters to our search criteria that don't affect things like categorical facets in your UI. Let's design another search page for our car dealer. This page will allow the user to search for a car and filter by color. Color choices are populated via an aggregation:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "query":{ "match": { "make": "ford" } },  
"post_filter": { <1> "term" : { "color" : "green" } }, "aggs" : { "all_colors": { "terms" : { "field" :  
"color" } } }
```



```
// SENSE: 300_Aggregations/45_filtering.json
```

<1> The `post_filter` element is a `+top-level+` element and filters just the search hits.

The `query` portion is finding all `+ford+` cars. We are then building a list of colors with a `terms` aggregation. Because aggregations operate in the query scope, the list of colors will correspond with the colors that Ford cars are painted.

Finally, the `post_filter` will filter the search results to show only green `+ford+` cars. This happens *after* the query is executed, so the aggregations are unaffected.

This is often important for coherent UIs. Imagine that a user clicks a category in your UI (for example, green). The expectation is that the search results are filtered, but *not* the UI options. If you applied a `filtered` query, the UI would instantly transform to show *only* `+green+` as an option--not what the user wants!

[WARNING]

Performance consideration

Use a `post_filter` *only* if you need to differentially filter search results and aggregations.
(((`"post filter", "performance and"`))) Sometimes people will use `post_filter` for regular searches.

Don't do this! The nature of the `post_filter` means it runs *after* the query, so any performance benefit of filtering (such as caches) is lost completely.

The `post_filter` should be used only in combination with aggregations, and only

when you need differential filtering.

[float="true"] === Recap

Choosing the appropriate type of filtering--search hits, aggregations, or both--often boils down to how you want your user interface to behave. Choose the appropriate filter (or combinations) depending on how you want to display results to your user.

- A `filtered` query affects both search results and aggregations.
- A `filter` bucket affects just aggregations.
- A `post_filter` affects just search results.



== Sorting Multivalue Buckets

Multivalue buckets—the `terms`, `histogram`, and `date_histogram`—dynamically produce many buckets.(((“sorting”, “of multivalue buckets”)))(((“buckets”, “multivalue, sorting”)))(((“aggregations”, “sorting multivalue buckets”))) How does Elasticsearch decide the order that these buckets are presented to the user?

By default, buckets are ordered by `doc_count` in(((“doc_count”, “buckets ordered by”))) descending order. This is a good default because often we want to find the documents that maximize some criteria: price, population, frequency. But sometimes you’ll want to modify this sort order, and there are a few ways to do it, depending on the bucket.

==== Intrinsic Sorts

These sort modes are *intrinsic* to the bucket: they operate on data that bucket(((“sorting”, “of multivalue buckets”, “intrinsic sorts”))) generates, such as `doc_count` .(((“buckets”, “multivalue, sorting”, “intrinsic sorts”))) They share the same syntax but differ slightly depending on the bucket being used.

Let’s perform a `terms` aggregation but sort by `doc_count`, in ascending order:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs" : { "colors" : { "terms" : { "field" : "color", "order": { "_count" : "asc" <1> } } } }
```

```
}
```

```
// SENSE: 300_Aggregations/50_sorting_ordering.json
```

<1> Using the `_count` keyword, we can sort by `doc_count`, in ascending order.

We introduce an `+order+` object(((“order parameter (aggregations)”))) into the aggregation, which allows us to sort on one of several values:

`_count` :: Sort by document count. Works with `terms`, `histogram`, `date_histogram`.

`_term` :: Sort by the string value of a term alphabetically. Works only with `terms`.

`_key` :: Sort by the numeric value of each bucket’s key (conceptually similar to `_term`). Works only with `histogram` and `date_histogram`.

==== Sorting by a Metric



Often, you'll find yourself wanting to sort based on a metric's calculated value.((("buckets", "multivalue", "sorting", "by a metric")))((("metrics", "sorting multivalue buckets by")))((("sorting", "of multivalue buckets", "sorting by a metric"))) For our car sales analytics dashboard, we may want to build a bar chart of sales by car color, but order the bars by the average price, ascending.

We can do this by adding a metric to our bucket, and then referencing that metric from the `+order+` parameter:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs" : { "colors" : { "terms" : { "field" : "color", "order": { "avg_price" : "asc" <2> } }, "aggs": { "avg_price": { "avg": { "field": "price" } <1> } } } }
```

```
}
```

```
// SENSE: 300_Aggregations/50_sorting_ordering.json
```

<1> The average price is calculated for each bucket.

<2> Then the buckets are ordered by the calculated average in ascending order.

This lets you override the sort order with any metric, simply by referencing the name of the metric. Some metrics, however, emit multiple values. The `extended_stats` metric is a good example: it provides half a dozen individual metrics.

If you want to sort on a multivalue metric,((("metrics", "sorting multivalue buckets by", "multivalue metric")))) you just need to use the dot-path to the metric of interest:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs" : { "colors" : { "terms" : { "field" : "color", "order": { "stats.variance" : "asc" <1> } }, "aggs": { "stats": { "extended_stats": { "field": "price" } } } } }
```

```
}
```

```
// SENSE: 300_Aggregations/50_sorting_ordering.json
```



<1> Using dot notation, we can sort on the metric we are interested in.

In this example we are sorting on the variance of each bucket, so that colors with the least variance in price will appear before those that have more variance.

==== Sorting Based on "Deep" Metrics

In the prior examples, the metric was a direct child of the bucket. An average price was calculated for each term.((("buckets", "multivalue", "sorting", "on deeper, nested metrics")))) ((("metrics", "sorting multivalue buckets by", "deeper, nested metrics")))) It is possible to sort on *deeper* metrics, which are grandchildren or great-grandchildren of the bucket--with some limitations.

You can define a path to a deeper, nested metric by using angle brackets (>), like so:

```
my_bucket>another_bucket>metric .
```

The caveat is that each nested bucket in the path must be a *single-value* bucket. A `filter` bucket produces((("filter bucket")))) a single bucket: all documents that match the filtering criteria. Multivalue buckets (such as `terms`) generate many dynamic buckets, which makes it impossible to specify a deterministic path.

Currently, there are only three single-value buckets: `filter` , `global` ((("global bucket"))), and `reverse_nested` . As a quick example, let's build a histogram of car prices, but order the buckets by the variance in price of red and green (but not blue) cars in each price range: ((("histograms", "buckets generated by, sorting on a deep metric"))))

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs" : { "colors" : { "histogram" : { "field" : "price", "interval": 20000, "order": { "red_green_cars>stats.variance" : "asc" <1> } }, "aggs": { "red_green_cars": { "filter": { "terms": { "color": ["red", "green"]}}}, <2> "aggs": { "stats": {"extended_stats": { "field" : "price"} } } } } }
```

```
}
```

```
// SENSE: 300_Aggregations/50_sorting_ordering.json
```

<1> Sort the buckets generated by the histogram according to the variance of a nested metric.

<2> Because we are using a single-value `filter` , we can use nested sorting.

<3> Sort on the stats generated by this metric.



In this example, you can see that we are accessing a nested metric. The `stats` metric is a child of `red_green_cars`, which is in turn a child of `colors`. To sort on that metric, we define the path as `red_green_cars>stats.variance`. This is allowed because the `filter` bucket is a single-value bucket.



== Approximate Aggregations

Life is easy if all your data fits on a single machine.((("aggregations", "approximate")))

Classic algorithms taught in CS201 will be sufficient for all your needs. But if all your data fits on a single machine, there would be no need for distributed software like Elasticsearch at all. But once you start distributing data, algorithm selection needs to be made carefully.

Some algorithms are amenable to distributed execution. All of the aggregations discussed thus far execute in a single pass and give exact results. These types of algorithms are often referred to as *embarrassingly parallel*, because they parallelize to multiple machines with little effort. When performing a `max` metric, for example, the underlying algorithm is very simple:

1. Broadcast the request to all shards.
2. Look at the `+price+` field for each document. If `price > current_max`, replace `current_max` with `price`.
3. Return the maximum price from all shards to the coordinating node.
4. Find the maximum price returned from all shards. This is the true maximum.

The algorithm scales linearly with machines because the algorithm requires no coordination (the machines don't need to discuss intermediate results), and the memory footprint is very small (a single integer representing the maximum).

Not all algorithms are as simple as taking the maximum value, unfortunately. More complex operations require algorithms that make conscious trade-offs in performance and memory utilization. There is a triangle of factors at play: big data, exactness, and real-time latency.

You get to choose two from this triangle:

Exact + real time: Your data fits in the RAM of a single machine. The world is your oyster; use any algorithm you want. Results will be 100% accurate and relatively fast.

Big data + exact: A classic Hadoop installation. Can handle petabytes of data and give you exact answers--but it may take a week to give you that answer.

Big data + real time: Approximate algorithms that give you accurate, but not exact, results.

Elasticsearch currently supports two approximate algorithms (`cardinality` and `percentiles`). ((("approximate algorithms")))((("cardinality")))((("percentiles")))) These will give you accurate results, but not 100% exact. In exchange for a little bit of estimation error, these algorithms give you fast execution and a small memory footprint.

For *most* domains, highly accurate results that return *in real time* across *all your data* is more important than 100% exactness. At first blush, this may be an alien concept to you. "We need exact answers!" you may yell. But consider the implications of a 0.5% error:



- The true 99th percentile of latency for your website is 132ms.
- An approximation with 0.5% error will be within +/- 0.66ms of 132ms.
- The approximation returns in milliseconds, while the "true" answer may take seconds, or be impossible.

For simply checking on your website's latency, do you care if the approximate answer is 132.66ms instead of 132ms? Certainly, not all domains can tolerate approximations--but the vast majority will have no problem. Accepting an approximate answer is more often a *cultural* hurdle rather than a business or technical imperative.



[[cardinality]] === Finding Distinct Counts

The first approximate aggregation provided by Elasticsearch is the `cardinality` metric.
((("cardinality", "finding distinct counts")))((("aggregations", "approximate", "cardinality")))
((("approximate algorithms", "cardinality")))((("distinct counts")))) This provides the cardinality
of a field, also called a *distinct* or *unique* count. ((("unique counts")))) You may be familiar
with the SQL version:

[source, sql]

```
SELECT DISTINCT(color)
```

FROM cars

Distinct counts are a common operation, and answer many fundamental business questions:

- How many unique visitors have come to my website?
- How many unique cars have we sold?
- How many distinct users purchased a product each month?

We can use the `cardinality` metric to determine the number of car colors being sold at our dealership:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs" : { "distinct_colors" : {  
"cardinality" : { "field" : "color" } } } }
```

```
}
```

```
// SENSE: 300_Aggregations/60_cardinality.json
```

This returns a minimal response showing that we have sold three different-colored cars:

[source,js]

```
... "aggregations": { "distinct_colors": { "value": 3 } }
```



...

We can make our example more useful: how many colors were sold each month? For that metric, we just nest the `cardinality` metric under `((("date histograms, building")))` a

```
date_histogram :
```

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs" : { "months" : {  
    "date_histogram": { "field": "sold", "interval": "month" }, "aggs": { "distinct_colors" : {  
        "cardinality" : { "field" : "color" } } } } }
```

```
}
```

```
// SENSE: 300_Aggregations/60_cardinality.json
```

===== Understanding the Trade-offs As mentioned at the top of this chapter, the `cardinality` metric is an approximate algorithm. `((("cardinality", "understanding the tradeoffs")))` It is based on the <http://bit.ly/1u6UWwd>[HyperLogLog++] (HLL) algorithm. `((("HLL (HyperLogLog) algorithm")))((("HyperLogLog (HLL) algorithm")))` HLL works by hashing your input and using the bits from the hash to make probabilistic estimations on the cardinality.

You don't need to understand the technical details (although if you're interested, the paper is a great read!), but you `((("memory usage", "cardinality metric")))` should be aware of the *properties* of the algorithm:

- Configurable precision, which controls memory usage (more precise == more memory).
- Excellent accuracy on low-cardinality sets.
- Fixed memory usage. Whether there are thousands or billions of unique values, memory usage depends on only the configured precision.

To configure the precision, you must specify the `precision_threshold` parameter. `((("precision_threshold parameter (cardinality metric))"))` This threshold defines the point under which cardinalities are expected to be very close to accurate. Consider this example:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs" : { "distinct_colors" : {  
    "cardinality" : { "field" : "color", "precision_threshold" : 100 <1> } } } }
```



```
// SENSE: 300_Aggregations/60_cardinality.json
```

<1> `precision_threshold` accepts a number from 0–40,000. Larger values are treated as equivalent to 40,000.

This example will ensure that fields with 100 or fewer distinct values will be extremely accurate. Although not guaranteed by the algorithm, if a cardinality is under the threshold, it is almost always 100% accurate. Cardinalities above this will begin to trade accuracy for memory savings, and a little error will creep into the metric.

For a given threshold, the HLL data-structure will use about `precision_threshold * 8` bytes of memory. So you must balance how much memory you are willing to sacrifice for additional accuracy.

Practically speaking, a threshold of `100` maintains an error under 5% even when counting millions of unique values.

==== Optimizing for Speed If you want a distinct count, you *usually* want to query your entire dataset (or nearly all of it). (((("cardinality", "optimizing for speed")))((("distinct counts", "optimizing for speed")))) Any operation on all your data needs to execute quickly, for obvious reasons. HyperLogLog is very fast already--it simply hashes your data and does some bit-twiddling.(((("HyperLogLog (HLL) algorithm")))((("HLL (HyperLogLog) algorithm"))))

But if speed is important to you, we can optimize it a little bit further. Since HLL simply needs the hash of the field, we can precompute that hash at index time.((("hashes, pre-computing for cardinality metric")))) When the query executes, we can skip the hash computation and load the value directly out of fielddata.

[NOTE]

Precomputing hashes is useful only on very large and/or high-cardinality fields. Calculating the hash on these fields is non-negligible at query time.

However, numeric fields hash very quickly, and storing the original numeric often requires the same (or less) memory. This is also true on low-cardinality string fields; there are internal optimizations that guarantee that hashes are calculated only once per unique value.

Basically, precomputing hashes is not guaranteed to make all fields faster -- only those that have high cardinality and/or large strings. And remember, precomputing simply shifts the cost to index time. You still pay the price;



you just choose *when* to pay it.

To do this, we need to add a new multfield to our data. We'll delete our index, add a new mapping that includes the hashed field, and then reindex:

[source,js]

```
DELETE /cars/
```

```
PUT /cars/ { "mappings": { "color": { "type": "string", "fields": { "hash": { "type": "murmur3" <1> } } } } }
```

```
POST /cars/transactions/_bulk { "index": {} } { "price" : 10000, "color" : "red", "make" : "honda", "sold" : "2014-10-28" } { "index": {} } { "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05" } { "index": {} } { "price" : 30000, "color" : "green", "make" : "ford", "sold" : "2014-05-18" } { "index": {} } { "price" : 15000, "color" : "blue", "make" : "toyota", "sold" : "2014-07-02" } { "index": {} } { "price" : 12000, "color" : "green", "make" : "toyota", "sold" : "2014-08-19" } { "index": {} } { "price" : 20000, "color" : "red", "make" : "honda", "sold" : "2014-11-05" } { "index": {} } { "price" : 80000, "color" : "red", "make" : "bmw", "sold" : "2014-01-01" } { "index": {} }
```

{ "price" : 25000, "color" : "blue", "make" : "ford", "sold" : "2014-02-12" }

```
// SENSE: 300_Aggregations/60_cardinality.json
```

<1> This multfield is of type `murmur3`, which is a hashing function.

Now when we run an aggregation, we use the `color.hash` field instead of the `color` field:

[source,js]

```
GET /cars/transactions/_search?search_type=count { "aggs": { "distinct_colors": { "cardinality": { "field": "color.hash" <1> } } } }
```

```
}
```

```
// SENSE: 300_Aggregations/60_cardinality.json
```



<1> Notice that we specify the hashed multifield, rather than the original.

Now the `cardinality` metric will load the values (the precomputed hashes) from `"color.hash"` and use those in place of dynamically hashing the original value.

The savings per document is small, but if hashing each field adds 10 nanoseconds and your aggregation touches 100 million documents, that adds 1 second per query. If you find yourself using `cardinality` across many documents, perform some profiling to see if precomputing hashes makes sense for your deployment.



[[percentiles]] === Calculating Percentiles

The other approximate metric offered by Elasticsearch is the `percentiles` metric.
((("percentiles")))(("aggregations", "approximate", "percentiles"))((("approximate algorithms", "percentiles")))) Percentiles show the point at which a certain percentage of observed values occur. For example, the 95th percentile is the value that is greater than 95% of the data.

Percentiles are often used to find outliers. In (statistically) normal distributions, the 0.13th and 99.87th percentiles represent three standard deviations from the mean. Any data that falls outside three standard deviations is often considered an anomaly because it is so different from the average value.

To be more concrete, imagine that you are running a large website and it is your job to guarantee fast response times to visitors. You must therefore monitor your website latency to determine whether you are meeting your goal.

A common metric to use in this scenario is the average latency. ((("metrics", "for website latency monitoring")))(("average metric")))) But this is a poor choice (despite being common), because averages can easily hide outliers. A median metric also suffers the same problem. ((("mean/median metric")))) You could try a maximum, but this metric is easily skewed by just a single outlier.

This graph in <> visualizes the problem. If you rely on simple metrics like mean or median, you might see a graph that looks like <>.

[[percentile-mean-median]] .Average request latency over time

image::images/elas_33in01.png["Assessing website latency using mean/median"]

Everything looks fine. ((("percentiles", "assessing website latency with")))) There is a slight bump, but nothing to be concerned about. But if we load up the 99th percentile (the value that accounts for the slowest 1% of latencies), we see an entirely different story, as shown in <>.

[[percentile-mean-median-percentile]] .Average request latency with 99th percentile over time

image::images/elas_33in02.png["Assessing website latency using percentiles"]

Whoa! At 9:30 a.m., the mean is only 75ms. As a system administrator, you wouldn't look at this value twice. Everything normal! But the 99th percentile is telling you that 1% of your customers are seeing latency in excess of 850ms--a very different story. There is also a smaller spike at 4:48 a.m. that wasn't even noticeable in the mean/median.

This is just one use-case for a percentile. Percentiles can also be used to quickly eyeball the distribution of data, check for skew or bimodalities, and more.



==== Percentile Metric

Let's load a new dataset (the car data isn't going to work well for percentiles). We are going to index a bunch of website latencies and run a few percentiles over it:

[source,js]

```
POST /website/logs/_bulk { "index": {} } { "latency" : 100, "zone" : "US", "timestamp" : "2014-10-28" } { "index": {} } { "latency" : 80, "zone" : "US", "timestamp" : "2014-10-29" } { "index": {} } { "latency" : 99, "zone" : "US", "timestamp" : "2014-10-29" } { "index": {} } { "latency" : 102, "zone" : "US", "timestamp" : "2014-10-28" } { "index": {} } { "latency" : 75, "zone" : "US", "timestamp" : "2014-10-28" } { "index": {} } { "latency" : 82, "zone" : "US", "timestamp" : "2014-10-29" } { "index": {} } { "latency" : 100, "zone" : "EU", "timestamp" : "2014-10-28" } { "index": {} } { "latency" : 280, "zone" : "EU", "timestamp" : "2014-10-29" } { "index": {} } { "latency" : 155, "zone" : "EU", "timestamp" : "2014-10-29" } { "index": {} } { "latency" : 623, "zone" : "EU", "timestamp" : "2014-10-28" } { "index": {} } { "latency" : 380, "zone" : "EU", "timestamp" : "2014-10-28" } { "index": {} }
```

```
{ "latency" : 319, "zone" : "EU", "timestamp" : "2014-10-29" }
```

// SENSE: 300_Aggregations/65_percentiles.json

This data contains three values: a latency, a data center zone, and a date timestamp. Let's run +percentiles+ over the whole dataset to get a feel for the distribution:

[source,js]

```
GET /website/logs/_search?search_type=count { "aggs" : { "load_times" : { "percentiles" : { "field" : "latency" <1> } }, "avg_load_time" : { "avg" : { "field" : "latency" <2> } } }
```

```
}
```

// SENSE: 300_Aggregations/65_percentiles.json

<1> The `percentiles` metric is applied to the `+latency+` field.

<2> For comparison, we also execute an `avg` metric on the same field.



By default, the `percentiles` metric will return an array of predefined percentiles: `[1, 5, 25, 50, 75, 95, 99]`. These represent common percentiles that people are interested in--the extreme percentiles at either end of the spectrum, and a few in the middle. In the response, we see that the fastest latency is around 75ms, while the slowest is almost 600ms. In contrast, the average is sitting near 200ms, which ((("average metric", "for website latency"))is much less informative:

[source,js]

```
... "aggregations": { "load_times": { "values": { "1.0": 75.55, "5.0": 77.75, "25.0": 94.75, "50.0": 101, "75.0": 289.75, "95.0": 489.3499999999985, "99.0": 596.2700000000002 } },  
"avg_load_time": { "value": 199.5833333333334 }
```

```
}
```

So there is clearly a wide distribution in latencies. Let's see whether it is correlated to the geographic zone of the data center:

[source,js]

```
GET /website/logs/_search?search_type=count { "aggs": { "zones": { "terms": { "field": "zone" <1> }, "aggs": { "load_times": { "percentiles": { <2> "field": "latency", "percents": [50, 95.0, 99.0] <3> } }, "load_avg": { "avg": { "field": "latency" } } } } }
```

```
}
```

// SENSE: 300_Aggregations/65_percentiles.json

<1> First we separate our latencies into buckets, depending on their zone.

<2> Then we calculate the percentiles per zone.

<3> The `+percents+` parameter accepts an array of percentiles that we want returned, since we are interested in only slow latencies.

From the response, we can see the EU zone is much slower than the US zone. On the US side, the 50th percentile is very close to the 99th percentile--and both are close to the average.



In contrast, the EU zone has a large difference between the 50th and 99th percentile. It is now obvious that the EU zone is dragging down the latency statistics, and we know that 50% of the EU zone is seeing 300ms+ latencies.

[source,js]

```
... "aggregations": { "zones": { "buckets": [ { "key": "eu", "doc_count": 6, "load_times": { "values": { "50.0": 299.5, "95.0": 562.25, "99.0": 610.85 } }, "load_avg": { "value": 309.5 } }, { "key": "us", "doc_count": 6, "load_times": { "values": { "50.0": 90.5, "95.0": 101.5, "99.0": 101.9 } }, "load_avg": { "value": 89.66666666666667 } } ] } }
```

...

==== Percentile Ranks

There is another, closely (((("approximate algorithms", "percentiles", "percentile ranks")))) (((("percentiles", "percentile ranks"))))related metric called `percentile_ranks`. The `percentiles` metric tells you the lowest value below which a given percentage of documents fall. For instance, if the 50th percentile is 119ms, then 50% of documents have values of no more than 119ms. The `percentile_ranks` tells you which percentile a specific value belongs to. The `percentile_ranks` of 119ms is the 50th percentile. It is basically a two-way relationship. For example:

- The 50th percentile is 119ms.
- The 119ms percentile rank is the 50th percentile.

So imagine that our website must maintain an SLA of 210ms response times or less. And, just for fun, your boss has threatened to fire you if response times creep over 800ms. Understandably, you would like to know what percentage of requests are actually meeting that SLA (and hopefully at least under 800ms!).

For this, you can apply the `percentile_ranks` metric instead of `percentiles`:

[source,js]

```
GET /website/logs/_search?search_type=count { "aggs" : { "zones" : { "terms" : { "field" : "zone" }, "aggs" : { "load_times" : { "percentile_ranks" : { "field" : "latency", "values" : [210, 800] <1> } } } } }
```



```
// SENSE: 300_Aggregations/65_percentiles.json
```

<1> The `percentile_ranks` metric accepts an array of values that you want ranks for.

After running this aggregation, we get two values back:

[source,js]

```
"aggregations": { "zones": { "buckets": [ { "key": "eu", "doc_count": 6, "load_times": { "values": { "210.0": 31.944444444444443, "800.0": 100 } } }, { "key": "us", "doc_count": 6, "load_times": { "values": { "210.0": 100, "800.0": 100 } } } ] }
```

```
}
```

This tells us three important things:

- In the EU zone, the percentile rank for 210ms is 31.94%.
- In the US zone, the percentile rank for 210ms is 100%.
- In both EU and US, the percentile rank for 800ms is 100%.

In plain english, this means that the EU zone is meeting the SLA only 32% of the time, while the US zone is always meeting the SLA. But luckily for you, both zones are under 800ms, so you won't be fired (yet!).

The `percentile_ranks` metric provides the same information as `percentiles`, but presented in a different format that may be more convenient if you are interested in specific value(s).

==== Understanding the Trade-offs

Like cardinality, calculating percentiles requires an approximate algorithm. The `naive(((("percentiles", "understanding the tradeoffs"))))((("approximate algorithms", "percentiles", "understanding the tradeoffs")))` implementation would maintain a sorted list of all values--but this clearly is not possible when you have billions of values distributed across dozens of nodes.

Instead, `percentiles` uses an algorithm called`((("TDigest algorithm")))` TDigest (introduced by Ted Dunning in <http://bit.ly/1DlpOWK> [Computing Extremely Accurate Quantiles Using T-Digests]). As with HyperLogLog, it isn't necessary to understand the full technical details, but it is good to know the properties of the algorithm:



- Percentile accuracy is proportional to how *extreme* the percentile is. This means that percentiles such as the 1st or 99th are more accurate than the 50th. This is just a property of how the data structure works, but it happens to be a nice property, because most people care about extreme percentiles.
- For small sets of values, percentiles are highly accurate. If the dataset is small enough, the percentiles may be 100% exact.
- As the quantity of values in a bucket grows, the algorithm begins to approximate the percentiles. It is effectively trading accuracy for memory savings. The exact level of inaccuracy is difficult to generalize, since it depends on your(("compression parameter (percentiles)")) data distribution and volume of data being aggregated.((("memory usage", "percentiles, controlling memory/accuracy ratio")))

Similar to `cardinality`, you can control the memory-to-accuracy ratio by changing a parameter: `compression`.

The TDigest algorithm uses nodes to approximate percentiles: the more nodes available, the higher the accuracy (and the larger the memory footprint) proportional to the volume of data. The compression parameter limits the maximum number of nodes to `20 * compression`.

Therefore, by increasing the compression value, you can increase the accuracy of your percentiles at the cost of more memory. Larger compression values also make the algorithm slower since the underlying tree data structure grows in size, resulting in more expensive operations. The default compression value is `100`.

A node uses roughly 32 bytes of memory, so in a worst-case scenario (for example, a large amount of data that arrives sorted and in order), the default settings will produce a TDigest roughly 64KB in size. In practice, data tends to be more random, and the TDigest will use less memory.



[[significant-terms]] == Significant Terms

The `significant_terms` (`SigTerms`) aggregation`(("significant_terms aggregation"))``((("aggregations", "Significant Terms")))` is rather different from the rest of the aggregations. All the aggregations we have seen so far are essentially simple math operations. By combining the various building blocks, you can build sophisticated aggregations and reports about your data.

`significant_terms` has a different agenda. To some, it may even look a bit like machine learning. `((("terms", "uncommonly common, finding with SigTerms aggregation")))` The `significant_terms` aggregation finds *uncommonly common* terms in your data-set.

What do we mean by *uncommonly common*? These are terms that are statistically unusual -- data that appears more frequently than the background rate would suggest. These statistical anomalies are usually indicative of something interesting in your data.

For example, imagine you are in charge of detecting and tracking down credit card fraud. Customers call and complain about unusual transactions appearing on their credit card -- their account has been compromised. These transactions are just symptoms of a larger problem. Somewhere in the recent past, a merchant has either knowingly stolen the customers' credit card information, or has unknowingly been compromised themselves.

Your job is to find the *common point of compromise*. If you have 100 customers complaining of unusual transactions, those customers likely share a single merchant--and it is this merchant that is likely the source of blame.

Of course, it is a little more nuanced than just finding a merchant that all customers share. For example, many of the customers will have large merchants like Amazon in their recent transaction history. We can rule out Amazon, however, since many uncompromised credit cards also have Amazon as a recent merchant.

This is an example of a *commonly common* merchant. Everyone, whether compromised or not, shares the merchant. This makes it of little interest to us.

On the opposite end of the spectrum, you have tiny merchants such as the corner drug store. These are *commonly uncommon*--only one or two customers have transactions from the merchant. We can rule these out as well. Since all of the compromised cards did not interact with the merchant, we can be sure it was not to blame for the security breach.

What we want are *uncommonly common* merchants. These are merchants that every compromised card shares, but that are not well represented in the background noise of uncompromised cards. These merchants are statistical anomalies; they appear more frequently than they should. It is highly likely that these uncommonly common merchants are to blame.



`significant_terms` aggregation does just this. It analyzes your data and finds terms that appear with a frequency that is statistically anomalous compared to the background data.

What you *do* with this statistical anomaly depends on the data. With the credit card data, you might be looking for fraud. With ecommerce, you might be looking for an unidentified demographic so you can market to them more efficiently. If you are analyzing logs, you might find one server that throws a certain type of error more often than it should. The applications of `significant_terms` is nearly endless.



==== significant_terms Demo

Because the `significant_terms` aggregation(((`"significant_terms aggregation"`, `"demonstration of"`))))((`"aggregations"`, `"significant_terms"`, `"demonstration of"`))) works by analyzing statistics, you need to have a certain threshold of data for it to become effective. That means we won't be able to index a small amount of example data for the demo.

Instead, we have a pre-prepared dataset of around 80,000 documents. This is saved as a snapshot (for more information about snapshots and restore, see <4>) in our public demo repository. You can "restore" this dataset into your cluster by using these commands:

[source,js]

```
PUT /_snapshot/sigterms <1> { "type": "url", "settings": { "url":  
"http://download.elasticsearch.org/definitiveguide/sigterms\_demo/" } }
```

```
GET /_snapshot/sigterms/_all <2>
```

```
POST /_snapshot/sigterms/snapshot/_restore <3>
```

GET /mlmovies,mlratings/_recovery <4>

```
// SENSE: 300_Aggregations/75_sigterms.json
```

<1> Register a new read-only URL repository pointing at the demo snapshot

<2> (Optional) Inspect the repository to learn details about available snapshots

<3> Begin the Restore process. This will download two indices into your cluster: `mlmovies` and `mlratings`

<4> (Optional) Monitor the Restore process using the Recovery API

NOTE: The dataset is around 50 MB and may take some time to download.

In this demo, we are going to look at movie ratings by users of MovieLens. At MovieLens, users make movie recommendations so other users can find new movies to watch. For this demo, we are going to recommend movies by using `significant_terms` based on an input movie.

Let's take a look at some sample data, to get a feel for what we are working with. There are two indices in this dataset, `mlmovies` and `mlratings`. Let's look at `mlmovies` first:



[source,js]

GET mlmovies/_search <1>

```
{ "took": 4, "timed_out": false, "_shards": {...}, "hits": { "total": 10681, "max_score": 1, "hits": [ { "_index": "mlmovies", "_type": "mlmovie", "_id": "2", "_score": 1, "_source": { "offset": 2, "bytes": 34, "title": "Jumanji (1995)" } },
```

....

// SENSE: 300_Aggregations/75_sigterms.json

<1> Execute a search without a query, so that we can see a random sampling of docs.

Each document in `mlmovies` represents a single movie. The two important pieces of data are the `_id` of the movie and the `title` of the movie. You can ignore `offset` and `bytes`; they are artifacts of the process used to extract this data from the original CSV files. There are 10,681 movies in this dataset.

Now let's look at `mlratings`:

[source,js]

GET mlratings/_search

```
{ "took": 3, "timed_out": false, "_shards": {...}, "hits": { "total": 69796, "max_score": 1, "hits": [ { "_index": "mlratings", "_type": "mlrating", "_id": "00IC-2jDQFiQkpD6vhbFYA", "_score": 1, "_source": { "offset": 1, "bytes": 108, "movie": [122,185,231,292,316,329,355,356,362,364,370,377,420,466,480,520,539,586,588,589,594,616], "user": 1 } },
```

...

// SENSE: 300_Aggregations/75_sigterms.json

Here we can see the recommendations of individual users. Each document represents a single user, denoted by the `user` ID field. The `movie` field holds a list of movies that this user watched and recommended.



==== Recommending Based on Popularity

The first strategy we could take is trying to recommend movies based on popularity. ((("popularity", "movie recommendations based on"))) Given a particular movie, we find all users who recommended that movie. Then we aggregate all their recommendations and take the top five most popular.

We can express that easily with a `terms` aggregation ((("terms aggregation", "movie recommendations (example)"))) and some filtering. Let's look at *Talladega Nights*, a comedy about NASCAR racing starring Will Ferrell. Ideally, our recommender should find other comedies in a similar style (and more than likely also starring Will Ferrell).

First we need to find the *Talladega Nights* ID:

[source,js]

```
GET mlmovies/_search { "query": { "match": { "title": "Talladega Nights" } } }
```

```
...
"hits": [
{
  "_index": "mlmovies",
  "_type": "mlmovie",
  "_id": "46970", <1>
  "_score": 3.658795,
  "_source": {
    "offset": 9575,
    "bytes": 74,
    "title": "Talladega Nights: The Ballad of Ricky Bobby (2006)"
  }
},
...
]
```

```
// SENSE: 300_Aggregations/75_sigterms.json
```

```
<1> Talladega Nights is ID 46970 .
```

Armed with the ID, we can now filter the ratings and ((("filtering", "in aggregations"))) apply our `terms` aggregation to find the most popular movies from people who also like *Talladega Nights*:

[source,js]



```
GET mlratings/_search?search_type=count <1> { "query": { "filtered": { "filter": { "term": { "movie": 46970 <2> } } } }, "aggs": { "most_popular": { "terms": { "field": "movie", <3> "size": 6 } } } }
```

```
}
```

// SENSE: 300_Aggregations/75_sigterms.json

<1> We execute our query on `mlratings` this time, and specify `search_type=count` since we are interested only in the aggregation results.

<2> Apply a filter on the ID corresponding to *Talladega Nights*.

<3> Finally, find the most popular movies by using a `terms` bucket.

We perform the search on the `mlratings` index, and apply a filter for the ID of *Talladega Nights*. Since aggregations operate on query scope, this will effectively filter the aggregation results to only the users who recommended *Talladega Nights*. Finally, we execute (({"terms aggregation", "movie recommendations (example)"})) a `terms` aggregation to bucket the most popular movies. We are requesting the top six results, since it is likely that *Talladega Nights* itself will be returned as a hit (and we don't want to recommend the same movie).

The results come back like so:

[source,js]

```
{ ... "aggregations": { "most_popular": { "buckets": [ { "key": 46970, "key_as_string": "46970", "doc_count": 271 }, { "key": 2571, "key_as_string": "2571", "doc_count": 197 }, { "key": 318, "key_as_string": "318", "doc_count": 196 }, { "key": 296, "key_as_string": "296", "doc_count": 183 }, { "key": 2959, "key_as_string": "2959", "doc_count": 183 }, { "key": 260, "key_as_string": "260", "doc_count": 90 } ] } }
```

...

We need to correlate these back to their original titles, which can be done with a simple filtered query:

[source,js]



```
GET mlmovies/_search { "query": { "filtered": { "filter": { "ids": { "values": [2571,318,296,2959,260] } } } }}
```

```
}
```

```
// SENSE: 300_Aggregations/75_sigterms.json
```

And finally, we end up with the following list:

1. Matrix, The
2. Shawshank Redemption
3. Pulp Fiction
4. Fight Club
5. Star Wars Episode IV: A New Hope

OK--well that is certainly a good list! I like all of those movies. But that's the problem: most **everyone** likes that list. Those movies are universally well-liked, which means they are popular on everyone's recommendations. The list is basically a recommendation of popular movies, not recommendations related to *Talladega Nights*.

This is easily verified by running the aggregation again, but without the filter on *Talladega Nights*. This will give a top-five most popular movie list:

[source,js]

```
GET mlratings/_search?search_type=count { "aggs": { "most_popular": { "terms": { "field": "movie", "size": 5 } } }}
```

```
}
```

```
// SENSE: 300_Aggregations/75_sigterms.json
```

This returns a list that is very similar:

1. Shawshank Redemption
2. Silence of the Lambs, The
3. Pulp Fiction
4. Forrest Gump
5. Star Wars Episode IV: A New Hope



Clearly, just checking the most popular movies is not sufficient to build a good, discriminating recommender.

==== Recommending Based on Statistics

Now that the scene is set, let's try using `significant_terms`. `significant_terms` will analyze the group of people who enjoy *Talladega Nights* (the *foreground* group) and determine what movies are most popular. (((("statistics, movie recommendations based on (example)"))) It will then construct a list of popular films for everyone (the *background* group) and compare the two.

The statistical anomalies will be the movies that are *over-represented* in the foreground compared to the background. Theoretically, this should be a list of comedies, since people who enjoy Will Ferrell comedies will recommend them at a higher rate than the background population of people.

Let's give it a shot:

[source,js]

```
GET mlratings/_search?search_type=count { "query": { "filtered": { "filter": { "term": { "movie": 46970 } } } }, "aggs": { "most_sig": { "significant_terms": { <1> "field": "movie", "size": 6 } } }
```

}

// SENSE: 300_Aggregations/75_sigterms.json

<1> The setup is nearly identical -- we just use `significant_terms` instead of `terms`.

As you can see, the query is nearly the same. We filter for users who liked *Talladega Nights*; this forms the foreground group. By default, `significant_terms` will use the entire index as the background, so we don't need to do anything special.

The results come back as a list of buckets similar to `terms`, but with some extra (((("buckets", "returned by significant_terms aggregation"))))metadata:

[source,js]

```
... "aggregations": { "most_sig": { "doc_count": 271, <1> "buckets": [ { "key": 46970, "key_as_string": "46970", "doc_count": 271, "score": 256.549815498155, "bg_count": 271 }, { "key": 52245, <2> "key_as_string": "52245", "doc_count": 59, <3> "score": 59 } ] }
```



```
17.66462367106966, "bg_count": 185 <4> }, { "key": 8641, "key_as_string": "8641",  
"doc_count": 107, "score": 13.884387742677438, "bg_count": 762 }, { "key": 58156,  
"key_as_string": "58156", "doc_count": 17, "score": 9.746428133759462, "bg_count": 28 }, {  
"key": 52973, "key_as_string": "52973", "doc_count": 95, "score": 9.65770100311672,  
"bg_count": 857 }, { "key": 35836, "key_as_string": "35836", "doc_count": 128, "score":  
9.199001116457955, "bg_count": 1610 } ]
```

...

<1> The top-level `doc_count` shows the number of docs in the foreground group.

<2> Each bucket lists the key (for example, movie ID) being aggregated.

<3> A `doc_count` for that bucket.

<4> And a background count, which shows the rate at which this value appears in the entire background.

You can see that the first bucket we get back is *Talladega Nights*. It is found in all 271 documents, which is not surprising. Let's look at the next bucket: key `52245`.

This ID corresponds to *Blades of Glory*, a comedy about male figure skating that also stars Will Ferrell. We can see that it was recommended 59 times by the people who also liked *Talladega Nights*. This means that 21% of the foreground group recommended *Blades of Glory* ($59 / 271 = 0.2177$).

In contrast, *Blades of Glory* was recommended only 185 times in the entire dataset, which equates to a mere 0.26% ($185 / 69796 = 0.00265$). *Blades of Glory* is therefore a statistical anomaly: it is uncommonly common in the group of people who like *Talladega Nights*. We just found a good recommendation!

If we look at the entire list, they are all comedies that would fit as good recommendations (many of which also star Will Ferrell):

1. *Blades of Glory*
2. *Anchorman: The Legend of Ron Burgundy*
3. *Semi-Pro*
4. *Knocked Up*
5. *40-Year-Old Virgin, The*

This is just one example of the power of `significant_terms`. Once you start using `significant_terms`, you find many situations where you don't want the most popular--you want the most uncommonly common. This simple aggregation can uncover some surprisingly sophisticated trends in your data.





[[fielddata]] === Fielddata

Aggregations work via a data structure known as *fielddata* (briefly introduced in <>).
((("fielddata")))((("memory usage", "fielddata")))Fielddata is often the largest consumer of memory in an Elasticsearch cluster, so it is important to understand how it works.

[TIP]

Fielddata can be loaded on the fly into memory, or built at index time and stored on disk.
((("fielddata", "loaded into memory vs. on disk"))) Later, we will talk about on-disk fielddata in <>. For now we will focus on in-memory fielddata, as it is currently the default mode of operation in Elasticsearch. This may well change in a future version.

=====

Fielddata exists because inverted indices are efficient only for certain operations. The inverted index excels((("inverted index", "fielddata versus"))) at finding documents that contain a term. It does not perform well in the opposite direction: determining which terms exist in a single document. Aggregations need this secondary access pattern.

Consider the following inverted index:

Term	Doc_1	Doc_2	Doc_3
<hr/>			
brown	X X		
dog	X X		
dogs	X X		
fox	X X		
foxes	X		
in	X		
jumped	X X		
lazy	X X		
leap	X		
over	X X X		
quick	X X X		
summer	X		
the	X X		
<hr/>			

If we want to compile a complete list of terms in any document that mentions +brown+, we might build a query like so:

[source,js]



```
GET /my_index/_search { "query" : { "match" : { "body" : "brown" } }, "aggs" : {  
  "popular_terms": { "terms" : { "field" : "body" } } }  
}
```

The query portion is easy and efficient. The inverted index is sorted by terms, so first we find `+brown+` in the terms list, and then scan across all the columns to see which documents contain `+brown+`. We can very quickly see that `Doc_1` and `Doc_2` contain the token `+brown+`.

Then, for the aggregation portion, we need to find all the unique terms in `Doc_1` and `Doc_2` .`((("aggregations", "fielddata", "using instead of inverted index")))` Trying to do this with the inverted index would be a very expensive process: we would have to iterate over every term in the index and collect tokens from `Doc_1` and `Doc_2` columns. This would be slow and scale poorly: as the number of terms and documents grows, so would the execution time.

Fielddata addresses this problem by inverting the relationship. While the inverted index maps terms to the documents containing the term, fielddata maps documents to the terms contained by the document:

Doc	Terms
<hr/>	
Doc_1	brown, dog, fox, jumped, lazy, over, quick, the
Doc_2	brown, dogs, foxes, in, lazy, leap, over, quick, summer
Doc_3	dog, dogs, fox, jumped, over, quick, the
<hr/>	

Once the data has been uninverted, it is trivial to collect the unique tokens from `Doc_1` and `Doc_2` . Go to the rows for each document, collect all the terms, and take the union of the two sets.

[TIP]

The fielddata cache is per segment.`((("fielddata cache")))((("segments", "fielddata cache")))` In other words, when a new segment becomes visible to search, the fielddata cached from old segments remains valid. Only the data for the new segment needs to be loaded into memory.



Thus, search and aggregations are closely intertwined. Search finds documents by using the inverted index. Aggregations collect and aggregate values from fielddata, which is itself generated from the inverted index.

The rest of this chapter covers various functionality that either decreases fielddata's memory footprint or increases execution speed.

[NOTE]

Fielddata is not just used for aggregations.((("fielddata", "uses other than aggregations"))) It is required for any operation that needs to look up the value contained in a specific document. Besides aggregations, this includes sorting, scripts that access field values, parent-child relationships (see <>), and certain types of queries or filters, such as the <> filter.

=====



[[aggregations-and-analysis]] === Aggregations and Analysis

Some aggregations, such as the `terms` bucket, operate((("analysis", "aggregations and"))(("aggregations", "and analysis"))) on string fields. And string fields may be either `analyzed` or `not_analyzed`, which begs the question: how does analysis affect aggregations? ((("strings", "analyzed or not_analyzed string fields"))(("not_analyzed fields"))(("analyzed fields")))

The answer is "a lot," but it is best shown through an example. First, index some documents representing various states in the US:

[source,js]

```
POST /agg_analysis/data/_bulk { "index": {} } { "state" : "New York" } { "index": {} } { "state" : "New Jersey" } { "index": {} } { "state" : "New Mexico" } { "index": {} } { "state" : "New York" } { "index": {} }
```

{ "state" : "New York" }

We want to build a list of unique states in our dataset, complete with counts. Simple--let's use a `terms` bucket:

[source,js]

```
GET /agg_analysis/data/_search?search_type=count { "aggs" : { "states" : { "terms" : { "field" : "state" } } }
```

```
}
```

This gives us these results:

[source,js]

```
{ ... "aggregations": { "states": { "buckets": [ { "key": "new", "doc_count": 5 }, { "key": "york", "doc_count": 3 }, { "key": "jersey", "doc_count": 1 }, { "key": "mexico", "doc_count": 1 } ] } }
```

```
}
```



Oh dear, that's not at all what we want! Instead of counting states, the aggregation is counting individual words. The underlying reason is simple: aggregations are built from the inverted index, and the inverted index is *post-analysis*.

When we added those documents to Elasticsearch, the string "New York" was analyzed/tokenized into ["new", "york"]. These individual tokens were then used to populate fielddata, and ultimately we see counts for new instead of New York.

This is obviously not the behavior that we wanted, but luckily it is easily corrected.

We need to define a multfield for +state+ and set it to not_analyzed. This will prevent New York from being analyzed, which means it will stay a single token in the aggregation. Let's try the whole process over, but this time specify a raw multifield:

[source,js]

```
DELETE /agg_analysis/ PUT /agg_analysis { "mappings": { "data": { "properties": { "state": { "type": "string", "fields": { "raw" : { "type": "string", "index": "not_analyzed" <1> } } } } } }
```

```
POST /agg_analysis/_bulk { "index": {} } { "state": "New York" } { "index": {} } { "state": "New Jersey" } { "index": {} } { "state": "New Mexico" } { "index": {} } { "state": "New York" } { "index": {} } { "state": "New York" }
```

```
GET /agg_analysis/_search?search_type=count { "aggs": { "states": { "terms": { "field": "state.raw" <2> } } } }
```

```
}
```

<1> This time we explicitly map the +state+ field and include a not_analyzed sub-field.

<2> The aggregation is run on +state.raw+ instead of +state+.

Now when we run our aggregation, we get results that make sense:

[source,js]

```
{ ... "aggregations": { "states": { "buckets": [ { "key": "New York", "doc_count": 3 }, { "key": "New Jersey", "doc_count": 1 }, { "key": "New Mexico", "doc_count": 1 } ] } }
```

```
}
```



In practice, this kind of problem is easy to spot. Your aggregations will simply return strange buckets, and you'll remember the analysis issue. It is a generalization, but there are not many instances where you want to use an analyzed field in an aggregation. When in doubt, add a multi-field so you have the option for both.((("analyzed fields", "aggregations and")))

===== High-Cardinality Memory Implications

There is another reason to avoid aggregating analyzed fields: high-cardinality fields consume a large amount of memory when loaded into fielddata.((("memory usage", "high-cardinality fields")))((("cardinality", "high-cardinality fields, memory use issues")))) The analysis process often (although not always) generates a large number of tokens, many of which are unique. This increases the overall cardinality of the field and contributes to more memory pressure.((("analysis", "high-cardinality fields, memory use issues"))))

Some types of analysis are *extremely* unfriendly with regards to memory. Consider an n-gram analysis process.((("n-grams", "memory use issues associated with")))) The term +New York+ might be n-grammed into the following tokens:

- ne
- ew
- +w{nbsp}+
- +{nbsp}y+
- yo
- or
- rk

You can imagine how the n-gramming process creates a huge number of unique tokens, especially when analyzing paragraphs of text. When these are loaded into memory, you can easily exhaust your heap space.

So, before aggregating across fields, take a second to verify that the fields are `not_analyzed`. And if you want to aggregate analyzed fields, ensure that the analysis process is not creating an obscene number of tokens.

[TIP]

At the end of the day, it doesn't matter whether a field is `analyzed` or `not_analyzed`. The more unique values in a field--the higher the cardinality of the field--the more memory that is required. This is especially true for string fields, where every unique string must be held in memory--longer strings use more memory.



地理坐标点



地理坐标点

地理坐标点 (*geo-point*) 是指地球表面可以用经纬度描述的一个点。地理坐标点可以用来计算两个坐标位置间的距离，或者判断一个点是否在一个区域中。

地理坐标点不能被动态映射 (dynamic mapping) 自动检测，而是需要显式声明对应字段类型为 `geo_point`。

```
PUT /attractions
{
  "mappings": {
    "restaurant": {
      "properties": {
        "name": {
          "type": "string"
        },
        "location": {
          "type": "geo_point"
        }
      }
    }
  }
}
```

经纬度坐标格式

如上例，`location` 被声明为 `geo_point` 后，我们就可以索引包含了经纬度信息的文档了。经纬度信息的形式可以是字符串，数组或者对象。



```
PUT /attractions/restaurant/1
{
  "name":      "Chipotle Mexican Grill",
  "location":  "40.715, -74.011" <1>
}

PUT /attractions/restaurant/2
{
  "name":      "Pala Pizza",
  "location":  { <2>
    "lat":       40.722,
    "lon":       -73.989
  }
}

PUT /attractions/restaurant/3
{
  "name":      "Mini Munchies Pizza",
  "location":  [ -73.983, 40.719 ] <3>
}
```

- <1> 以半角逗号分割的字符串形式 `"lat,lon"`；
- <2> 明确以 `lat` 和 `lon` 作为属性的对象；
- <3> 数组形式表示 `[lon,lat]`。

注意

可能所有人都至少踩过一次这个坑：地理坐标点用字符串形式表示时是纬度在前，经度在后（`"latitude,longitude"`），而数组形式表示时刚好相反，是经度在前，纬度在后（`[longitude,latitude]`）。

其实，在 Elasticsearch 内部，不管字符串形式还是数组形式，都是纬度在前，经度在后。不过早期为了适配 GeoJSON 的格式规范，调整了数组形式的表示方式。

因此，在使用地理位置（geolocation）的路上就出现了这么一个“捕熊器”，专坑那些不了解这个陷阱的使用者。



通过地理坐标点过滤

有四种地理坐标点相关的过滤方式可以用来选中或者排除文档：

- `geo_bounding_box` ::

找出落在指定矩形框中的坐标点

- `geo_distance` ::

找出与指定位置在给定距离内的点

- `geo_distance_range` ::

找出与指定点距离在给定最小距离和最大距离之间的点

- `geo_polygon` ::

找出落在多边形中的点。这个过滤器使用代价很大。当你觉得自己需要使用它，最好先看看 [geo-shapes](#)

所有这些过滤器的工作方式都相似：把索引中所有文档（而不仅仅是查询中匹配到的部分文档，见 [fielddata-intro](#)）的经纬度信息都载入内存，然后每个过滤器执行一个轻量级的计算去判断当前点是否落在指定区域。

提示

地理坐标过滤器使用代价昂贵——所以最好在文档集合尽可能少的场景使用。你可以先使用那些简单快捷的过滤器，比如 `term` 或者 `range`，来过滤掉尽可能多的文档，最后才交给地理坐标过滤器处理。

布尔型过滤器（`bool filter`）会自动帮你做这件事。它会优先让那些基于“bitset”的简单过滤器（见 [filter-caching](#)）来过滤掉尽可能多的文档，然后依次才是地理坐标过滤器或者脚本类的过滤器。



地理坐标盒模型过滤器

这是目前为止最有效的 地理坐标过滤器了，因为它计算起来非常简单。你指定一个矩形的顶部（`top`），底部（`bottom`），左边界（`left`），和 右边界（`right`），然后它只需判断坐标的经度是否在左右边界之间，纬度是否在上下边界之间。（译注：原文似乎写反了）

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "location": { <1>
            "top_left": {
              "lat": 40.8,
              "lon": -74.0
            },
            "bottom_right": {
              "lat": 40.7,
              "lon": -73.0
            }
          }
        }
      }
    }
  }
}
```

- <1> 盒模型信息也可以用 `bottom_left`（左下方点）和 `top_right`（右上方点）来表示。

优化盒模型

地理坐标盒模型过滤器 不需要把所有坐标点都加载到内存里。因为它要做的只是简单判断 纬度 和 经度 坐标数值是否在给定的范围内，所以它可以用倒排索引来做一个范围（`range`）过滤。

要使用这种优化方式，需要把 `geo_point` 字段用 纬度（`lat`）和 经度（`lon`）方式表示并分别索引。



```
PUT /attractions
{
  "mappings": {
    "restaurant": {
      "properties": {
        "name": {
          "type": "string"
        },
        "location": {
          "type": "geo_point",
          "lat_lon": true <1>
        }
      }
    }
  }
}
```

- <1> `location.lat` 和 `location.lon` 字段将被分别索引。它们可以被用于检索，但是不会在检索结果中返回。

然后，查询时你需要告诉 Elasticsearch 使用已索引的 `lat` 和 `lon`。

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "type": "indexed", <1>
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.0
            },
            "bottom_right": {
              "lat": 40.7,
              "lon": -73.0
            }
          }
        }
      }
    }
  }
}
```

- <1> 设置 `type` 参数为 `indexed` (默认为 `memory`) 来明确告诉 Elasticsearch 对这个过滤器使用倒排索引。



注意：

`geo_point` 类型可以包含多个地理坐标点，但是针对经度纬度分别索引的这种优化方式（`lat_lon`）只对单个坐标点的方式有效。



地理距离过滤器

地理距离过滤器（`geo_distance`）以给定位置为圆心画一个圆，来找出那些位置落在其中的文档：

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_distance": {
          "distance": "1km", <1>
          "location": { <2>
            "lat": 40.715,
            "lon": -73.988
          }
        }
      }
    }
  }
}
```

- <1> 找出所有与指定点距离在1公里（`1km`）内的 `location` 字段。访问 [Distance Units](#) 查看所支持的距离表示单位
- <2> 中心点可以表示为字符串，数组或者（如示例中的）对象。详见 [lat-lon-formats](#)。

地理距离过滤器计算代价昂贵。为了优化性能，Elasticsearch 先画一个矩形框（边长为2倍距离）来围住整个圆形，这样就可以用消耗较少的盒模型计算方式来排除掉那些不在盒子内（自然也不在圆形内）的文档，然后只对落在盒模型内的这部分点用地理坐标计算方式处理。

提示

你需要判断你的使用场景，是否需要如此精确的使用圆模型来做距离过滤？通常使用矩形模型是更高效的方式，并且往往也能满足应用需求。

更快的地理距离计算

两点间的距离计算，有多种性能换精度的算法：

- `arc ::`



最慢但是最精确是 弧形（arc）计算方式，这种方式把世界当作是球体来处理。不过这种方式精度还是有限，因为这个世界并不是完全的球体。

- `plane ::`

平面（plane）计算方式，(((“plane distance calculation”)))把地球当成是平坦的。这种方式快一些但是精度略逊；在赤道附近位置精度最好，而靠近两极则变差。

- `sloppy_arc ::`

如此命名，是因为它使用了 Lucene 的 `sloppyMath` 类。这是一种用精度换取速度的计算方式，它使用 [Haversine formula](#) 来计算距离；它比 弧形（arc）计算方式快4~5倍，并且距离精度达99.9%。这也是默认的计算方式。

你可以参考下例来指定不同的计算方式：

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_distance": {
          "distance": "1km",
          "distance_type": "plane", <1>
          "location": {
            "lat": 40.715,
            "lon": -73.988
          }
        }
      }
    }
  }
}
```

- <1> 使用更快但精度稍差的 平面（plane）计算方式。

提示：你的用户真的会在意一个宾馆落在指定圆形区域数米之外了吗？一些地理位置相关的应用会有较高的精度要求；但大部分实际应用场景中，使用精度较低但响应更快的计算方式可能就挺好。

地理距离区间过滤器

地理距离过滤器（`geo_distance`）和 地理距离区间过滤器（`geo_distance_range`）的唯一差别在于后者是一个环状的，它会排除掉落在内圈中的那部分文档。

指定到中心点的距离也可以换一种表示方式：指定一个最小距离（使用 `gt` 或者 `gte`）和最大距离（使用 `lt` 或者 `lte`），就像使用 区间（`range`）过滤器一样。



```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_distance_range": {
          "gte": "1km", <1>
          "lt": "2km", <1>
          "location": {
            "lat": 40.715,
            "lon": -73.988
          }
        }
      }
    }
  }
}
```

- <1> 匹配那些距离中心点超过 1公里 而小于 2公里 的位置。



缓存地理位置过滤器

因为如下两个原因，地理位置过滤器默认是不被缓存的：

- 地理位置过滤器通常是用于查找用户当前位置附近的东西。但是用户是在移动的，并且没有两个用户的位置完全相同，因此缓存的过滤器基本不会被重复使用到。
- 过滤器是被缓存为比特位集合来表示段（`segment`）内的文档。假如我们的查询排除了几乎所有文档，只剩一个保存在这个特别的段内。一个未缓存的地理位置过滤器只需要检查这一个文档就行了，但是一个缓存的地理位置过滤器则需要检查所有在段内的文档。

缓存对于地理位置过滤器也可以很有效。假设你的索引里包含了所有美国的宾馆。一个在纽约的用户是不会对旧金山的宾馆感兴趣的。所以我们可以认为纽约是一个热点（*hot spot*），然后画一个边框把它和附近的区域围起来。

如果这个 地理盒模型过滤器（`geo_bounding_box`）被缓存起来，那么当有位于纽约市的用户访问时它就可以被重复使用了。它可以直接排除国内其它区域的宾馆。然后我们使用未缓存的，更加明确的 地理盒模型过滤器（`geo_bounding_box`）或者 地理距离过滤器（`geo_distance`）来在剩下的结果集中把范围进一步缩小到用户附近：



```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "bool": {
          "must": [
            {
              "geo_bounding_box": {
                "type": "indexed",
                "_cache": true, <1>
                "location": {
                  "top_left": {
                    "lat": 40.8,
                    "lon": -74.1
                  },
                  "bottom_right": {
                    "lat": 40.4,
                    "lon": -73.7
                  }
                }
              }
            },
            {
              "geo_distance": { <2>
                "distance": "1km",
                "location": {
                  "lat": 40.715,
                  "lon": -73.988
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

- <1> 缓存的 地理盒模型过滤器 把结果集缩小到了纽约市。
- <2> 代价更高的 地理距离过滤器（ geo_distance ）让结果集缩小到1km内的用户。



减少内存占用

每一个 经纬度（lat/lon）组合需要占用16个字节的内存。要知道内存可是供不应求的。使用这种占用16字节内存的方式可以得到非常精确的结果。不过就像之前提到的一样，实际应用中几乎都不需要这么精确。

你可以通过这种方式来减少内存使用量：设置一个 压缩的（compressed）数据字段格式并明确指定你的地理坐标点所需的精度。即使只是将精度降低到1毫米（1mm）级别，也可以减少1/3的内存使用。更实际的，将精度设置到3米（3m）内存占用可以减少62%，而设置到1公里（1km）则节省75%之多。

这个设置项可以通过 `update-mapping API` 来对实时索引进行调整：

```
POST /attractions/_mapping/restaurant
{
  "location": {
    "type": "geo_point",
    "fielddata": {
      "format": "compressed",
      "precision": "1km" <1>
    }
  }
}
```

- <1> 每一个 经纬度（lat/lon）组合现在只需要4个字节，而不是16个。

另外，你还可以这样做来避免把所有地理坐标点全部同时加载到内存中：使用在优化盒模型（`optimize-bounding-box`）中提到的技术，或者把地理坐标点当作文档值（`doc values`）来存储。



```
PUT /attractions
{
  "mappings": {
    "restaurant": {
      "properties": {
        "name": {
          "type": "string"
        },
        "location": {
          "type": "geo_point",
          "doc_values": true <1>
        }
      }
    }
  }
}
```

- <1> 地理坐标点现在不会被加载到内存，而是保存在磁盘中。

将地理坐标点映射为文档值的方式只能是在这个字段第一次被创建时。相比使用字段值，使用文档值会有一些小的性能代价，不过考虑到它对内存的节省，这种方式通常是还值得的。



按距离排序

检索结果可以按跟指定点的距离排序：

提示 当你可以（*can*）按距离排序时，按距离打分（**scoring-by-distance**）通常是一个更好的解决方案。

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "type": "indexed",
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.0
            },
            "bottom_right": {
              "lat": 40.4,
              "lon": -73.0
            }
          }
        }
      }
    },
    "sort": [
      {
        "_geo_distance": {
          "location": { <1> },
          "lat": 40.715,
          "lon": -73.998
        },
        "order": "asc",
        "unit": "km", <2>
        "distance_type": "plane" <3>
      }
    ]
  }
}
```

- <1> 计算每个文档中 `location` 字段与指定的 `lat/lon` 点间的距离。
- <2> 以 公里（`km`）为单位，将距离设置到每个返回结果的 `sort` 键中。
- <3> 使用快速但精度略差的 平面（`plane`）计算方式。



你可能想问：为什么要制定距离的 单位（unit）呢？用于排序的话，我们并不关心比较距离的尺度是英里，公里还是光年。原因是，这个用于排序的值会设置在每个返回结果的 `sort` 元素中。

```
...
  "hits": [
    {
      "_index": "attractions",
      "_type": "restaurant",
      "_id": "2",
      "_score": null,
      "_source": {
        "name": "New Malaysia",
        "location": {
          "lat": 40.715,
          "lon": -73.997
        }
      },
      "sort": [
        0.08425653647614346 <1>
      ]
    },
    ...
  ]
```

- <1> 宾馆距离我们的指定位置距离是 0.084km。
- 你可以通过设置 单位（unit）来让返回值的形式跟你应用中想要的匹配。

提示

地理距离排序可以对多个坐标点来使用，不管（这些坐标点）是在文档中还是排序参数中。使用 `sort_mode` 来指定是否需要使用位置集合的 最小（`min`），最大（`max`）或者 平均（`avg`）距离。这样就可以返回 离我的工作地和家最近的朋友 这样的结果了。

按距离打分

有可能距离只是决定返回结果排序的唯一重要因素，不过更常见的情况是距离会和其它因素，比如全文检索匹配度，流行程度或者价格一起决定排序结果。

遇到这种场景你需要在查询分值计算（`function_score query`）中指定方式让我们把这些因子处理得到一个综合分。`decay-functions` 中有个一个例子就是地理距离影响排序得分的。

另外按距离排序还有个缺点就是性能：需要对每一个匹配到的文档都进行距离计算。而 `function_score` 请求，在 `rescore phase` 阶段有可能只需要对前 n 个结果进行计算处理。



Geohashes



Geohashes

Geohashes 是一种将 经纬度坐标对（lat/lon）编码成字符串的方式。最开始这么做只是为了让地理位置在url上呈现的形式更加友好，不过现在geohash已经变成一种在数据库中有有效索引地理坐标点和地理形状的方式。

Geohashes 把整个世界分为32个单元的格子--4行8列--每一个格子都用一个字母或者数字标识。比如 g 这个单元覆盖了半个格林兰，冰岛的全部和大不列颠的大部分。每一个单元还可以进一步被分解成新的32个单元，这些单元又可以继续被分解成32个更小的单元，不断重复下去。 gc 这个单元覆盖了爱尔兰和英格兰， gcp 覆盖了伦敦的大部分和部分南英格兰， gcpuuz94k 是伯明翰宫的入口，精确到了约5米。

换句话说，geohash的长度越长，它的精度就越高。如果两个geohash有一个共同的前缀，如 gcpuuz ，就表示他们挨得很紧。共同的前缀越长，距离就越近。

但那也就是说，两个刚好相邻的位置，会可能有完全不同的geohash。一个实例，伦敦的 **Millenium Dome** 的geohash是 u10hbp ，因为它落在了 u 这个大单元里，而紧挨着它东边的最大的单元是 g 。

地理坐标点可以自动关联到他们对应的 geohash 。需要注意的是，他们会被索引到了所有（各个层级）的 geohash 前缀（ prefixes ）。例：索引伯明翰宫的门口--坐标纬度 51.501568 ，经度 -0.141257 --会在各种尺寸精度的 geohash 上建立索引，如下表：

Geohash	Level	Dimensions
g	1	~ 5,004km x 5,004km
gc	2	~ 1,251km x 625km
gcp	3	~ 156km x 156km
gcpu	4	~ 39km x 19.5km
gcpuu	5	~ 4.9km x 4.9km
gcpuuz	6	~ 1.2km x 0.61km
gcpuuz9	7	~ 152.8m x 152.8m
gcpuuz94	8	~ 38.2m x 19.1m
gcpuuz94k	9	~ 4.78m x 4.78m
gcpuuz94kk	10	~ 1.19m x 0.60m
gcpuuz94kkp	11	~ 14.9cm x 14.9cm
gcpuuz94kkp5	12	~ 3.7cm x 1.8cm



geohash单元过滤器（`geohash_cell filter`）可以使用这些geohash前缀来找出与指定坐标点（`lat/lon`）相邻的位置。



Geohashes 映射

首先，你需要确定你需要什么样的精度。虽然你也可以使用12级的精度来索引所有的地理坐标点，但是你真的需要精确到数厘米的精度吗？如果你把精度控制在一个实际一些的值，比如 `1km`，那么你可以节省大量的索引空间：

```
PUT /attractions
{
  "mappings": {
    "restaurant": {
      "properties": {
        "name": {
          "type": "string"
        },
        "location": {
          "type": "geo_point",
          "geohash_prefix": true, <1>
          "geohash_precision": "1km" <2>
        }
      }
    }
  }
}
```

- <1> 将 `geohash_prefix` 设为 `true` 来告诉 Elasticsearch 使用指定精度来做 geohash 前缀索引。
- <2> 精度描述可以是一个具体数字，表示 geohash 的长度，也可以是一个距离。精度设为 `1km` 表示 geohash 长度是 7。

通过如上设置，geohash前缀为 1-7 的部分将被索引，所能提供的精度大约在150米。



geohash单元过滤器

geohash单元 过滤器做的事情非常简单：把经纬度坐标位置根据指定精度转换成一个 geohash，然后查找落在同一个geohash中的位置--这实在是非常高效的过滤器。

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geohash_cell": {
          "location": {
            "lat": 40.718,
            "lon": -73.983
          },
          "precision": "2km" <1>
        }
      }
    }
  }
}
```

- <1> precision 字段设置的精度不能高于geohash精度映射时的设定。

这个过滤器将坐标点转换成对应长度的geohash--本例中为 dr5rsk --然后查找位于同一个组中的所有位置。

然而，如上例中的写法可能不会返回5km内所有的宾馆。要知道每个 geohash 实际上仅是一个矩形，而指定的点可能位于这个矩形中的任何位置。有可能这个点刚好落在了geohash单元的边缘附近，但过滤器会排除那些（挨得很近却）落在相邻单元里的宾馆。

为了修正这点，我们可以告诉过滤器，把周围的单元也包含进来。通过设置 neighbors 参数为 true：



```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geohash_cell": {
          "location": {
            "lat": 40.718,
            "lon": -73.983
          },
          "neighbors": true, <1>
          "precision": "2km"
        }
      }
    }
  }
}
```

- <1> 过滤器将会查找对应的geohash和包围它的（8个）geohash。

明显的，`2km` 精度的geohash再加上周围的单元，会导致结果实际在一个更大的检索范围。这个过滤器不是为精度而生的，但是它非常有效率，可以用于更高精度的地理位置过滤器的前置过滤器。

提示

将 `precision` 参数设置为一个距离可能会有误导性。比如将 `precision` 设置为 `2km` 将会转换成长度为6的geohash。但实际上它的尺寸是约 $1.2\text{km} * 0.6 \text{ km}$ 。你可能会发现这还不如自己明确的设置一个长度 `5` 或者 `6` 来得更容易理解。

这个过滤器有一个比 地理盒模型过滤器 (`geo_bounding_box`) 更好的优点，就是它支持一个字段中有多个坐标位置的情况。我们在设置优化盒模型过滤器 (`optimize-bounding-box`) 讲过，设置 `lat_lon` 选项也是一个很有效的方式，但是它只对字段中的单个坐标点情况有效。



地理位置聚合



地理位置聚合

虽然地理位置过滤或评分功能很有用，不过更有用得是将信息再地图上呈现给用户。检索的结果集可能很多而不能将每个点都一一呈现，这时候就可以使用地理位置聚合来把这些位置点分布到更加可控的桶（**buckets**）里。

有三种聚合器可以作用于 `geo_point` 类型的字段：

- `geo_distance`

将文档按以指定中心点为圆心的圆环分组

- `geohash_grid`

将文档按 geohash 单元分组，以便在地图上呈现

- `geo_bounds`

返回包含一系列矩形框的经纬坐标对，这些矩形框包含了所有的坐标点。这种方式对于要在地图上选择一个合适的缩放等级（**zoom level**）时很实用。



按距离聚合

按距离聚合对于类似“找出距我1公里内的所有pizza店”这样的检索场景很适合。检索结果需要确实地只返回距离用户1km内的文档，不过我们可以再加上一个“1-2km内的结果集”：

```
GET /attractions/restaurant/_search
{
  "query": {
    "filtered": {
      "query": {
        "match": { <1>
          "name": "pizza"
        }
      },
      "filter": {
        "geo_bounding_box": {
          "location": { <2>
            "top_left": {
              "lat": 40.8,
              "lon": -74.1
            },
            "bottom_right": {
              "lat": 40.4,
              "lon": -73.7
            }
          }
        }
      }
    },
    "aggs": {
      "per_ring": {
        "geo_distance": { <3>
          "field": "location",
          "unit": "km",
          "origin": {
            "lat": 40.712,
            "lon": -73.988
          },
          "ranges": [
            { "from": 0, "to": 1 },
            { "from": 1, "to": 2 }
          ]
        }
      }
    },
    "post_filter": { <4>
      "geo_distance": {
        "distance": "1km"
      }
    }
  }
}
```



```
"distance": "1km",
"location": {
    "lat": 40.712,
    "lon": -73.988
}
}
}
}
```

- <1> 主查询查找饭店名中包含了“pizza”的文档。
- <2> 矩形框过滤器让结果集缩小到纽约区域。
- <3> 距离聚合器计算距用户1km和1km-2km的结果数。
- <4> 最后，后置过滤器（`post_filter`）再把结果缩小到距离用户1km的饭店。

上例请求的返回结果如下：



```
"hits": {
    "total":      1,
    "max_score": 0.15342641,
    "hits": [ <1>
        {
            "_index": "attractions",
            "_type":  "restaurant",
            "_id":    "3",
            "_score": 0.15342641,
            "_source": {
                "name": "Mini Munchies Pizza",
                "location": [
                    -73.983,
                    40.719
                ]
            }
        }
    ],
    "aggregations": {
        "per_ring": { <2>
            "buckets": [
                {
                    "key":      "*-1.0",
                    "from":     0,
                    "to":       1,
                    "doc_count": 1
                },
                {
                    "key":      "1.0-2.0",
                    "from":     1,
                    "to":       2,
                    "doc_count": 1
                }
            ]
        }
    }
}
```

- <1> 后置过滤器（`post_filter`）已经结果集缩小到满足“距离用户 1km”条件下的唯一一个pizza店。
- <2> 聚合器包含了“距离用户 2km”的pizza店的检索结果。

这个例子中，我们统计了落到各个环形区域中的饭店数。当然，我们也可以使用子聚合器再在每个环形区域中进一步计算它们的平均价格，最流行，等等。



geohash单元聚合器

一个查询返回的结果集中可能包含很多的点，以至于不能在地图上全部单独显示。 geohash 单元聚合器可以按照你指定的精度计算每个点的geohash并将相邻的点聚合到一起。

返回结果是一个个单元格，每个单元格对应一个可以在地图上展示的 geohash。通过改变 geohash 的精度，你可以统计全球、某个国家，或者一个城市级别的综述信息。

聚合结果是稀疏 (*sparse*) 的，因为它只返回包含了文档集合的单元。如果你的geohash精度太细，导致生成了太多的结果集，它默认只会返回包含结果最多的10000个单元 -- 它们包含了大部分文档集合。然后，为了找出这排在前10000的单元，它还是需要先生成所有的结果集。你可以通过如下方式控制生成的单元的数目：

- 1. 使用一个矩形过滤器来限制结果集。
- 1. 对该矩形，选择一个合适的精度。



```
GET /attractions/restaurant/_search?search_type=count
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "location": { <1>
            "top_left": {
              "lat": 40.8,
              "lon": -74.1
            },
            "bottom_right": {
              "lat": 40.4,
              "lon": -73.7
            }
          }
        }
      }
    },
    "aggs": {
      "new_york": {
        "geohash_grid": { <2>
          "field": "location",
          "precision": 5
        }
      }
    }
  }
}
```

- <1> 矩形框将检索限制在纽约区域。
- <2> 使用精度为 5 的 geohash，精度大约是 5km x 5km.

每个精度为 5 的 geohash 覆盖约 25 平方公里，那 10000 个单元就能覆盖 25 万平方公里。我们指定的矩形框覆盖面积约 $44\text{km} \times 33\text{km}$ ，也就是大概 1452 平方公里。所以这肯定在一个安全的限度内，我们不会因此浪费大量内存来生成太多单元。

上例请求的返回如下：



```
...
"aggregations": {
  "new_york": {
    "buckets": [ <1>
      {
        "key": "dr5rs",
        "doc_count": 2
      },
      {
        "key": "dr5re",
        "doc_count": 1
      }
    ]
  }
}
...
}
```

- <1> 每个单元以一个 geohash 作为 key 。

Again, we didn't specify any subaggregations, so all we got back was the document count. We could have asked for popular restaurant types, average price, or other details. 同样的，我们没有指定子聚合器，所以我们的返回结果是文档数目。我们也可以（指定子聚合器来）得到流行的饭店类型，平均价格，或者其它详细信息。

提示

为了将这些单元放置在地图上展示，我们需要一个类库来将geohash解析为对于的矩形框或者中心点。Javascript和一些语言中有现成的类库，不过你也可以根据 [geo-bounds-agg](#) 的信息自己来实现。



范围（边界）聚合器

在geohash聚合器的例子中，我们使用了一个矩形框过滤器来将结果限制在纽约区域。然而，我们的结果都分布在曼哈顿。当在地图上呈现给用户时，合理的方式是可以缩放到有数据的区域；地图上有大量空白区域是没有任何点分布的。

范围过滤器是这么做得：它计算出一个个小矩形框来覆盖到所有的坐标点。

```
GET /attractions/restaurant/_search?search_type=count
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.1
            },
            "bottom_right": {
              "lat": 40.4,
              "lon": -73.9
            }
          }
        }
      }
    }
  },
  "aggs": {
    "new_york": {
      "geohash_grid": {
        "field": "location",
        "precision": 5
      }
    },
    "map_zoom": { <1>
      "geo_bounds": {
        "field": "location"
      }
    }
  }
}
```

- <1> 范围聚合器会计算出一个最小的矩形框来覆盖查询结果的所有文档。

返回结果包含了一个可以用来在地图上缩放的矩形框：



```
...
"aggregations": {
  "map_zoom": {
    "bounds": {
      "top_left": {
        "lat": 40.722,
        "lon": -74.011
      },
      "bottom_right": {
        "lat": 40.715,
        "lon": -73.983
      }
    }
  },
  ...
}
```

实际上，我们可以把矩形聚合器放到每一个 geohash 单元里，因为有坐标点的单元只占了所有单元的一部分：



```
GET /attractions/restaurant/_search?search_type=count
{
  "query": {
    "filtered": {
      "filter": {
        "geo_bounding_box": {
          "location": {
            "top_left": {
              "lat": 40.8,
              "lon": -74.1
            },
            "bottom_right": {
              "lat": 40.4,
              "lon": -73.9
            }
          }
        }
      }
    }
  },
  "aggs": {
    "new_york": {
      "geohash_grid": {
        "field": "location",
        "precision": 5
      },
      "aggs": {
        "cell": { <1>
          "geo_bounds": {
            "field": "location"
          }
        }
      }
    }
  }
}
```

- <1> 子聚合器 `cell_bounds` 会作用于每个 `geohash` 单元。

现在落在每个`geohash`单元中的点都有了一个所在的矩形框区域：



```
...
"aggregations": {
  "new_york": {
    "buckets": [
      {
        "key": "dr5rs",
        "doc_count": 2,
        "cell": {
          "bounds": {
            "top_left": {
              "lat": 40.722,
              "lon": -73.989
            },
            "bottom_right": {
              "lat": 40.719,
              "lon": -73.983
            }
          }
        }
      },
      ...
    ],
    ...
  }
}
```



地理形状



地理形状

地理形状 (**geo-shapes**) 使用一种与地理坐标点完全不同的方法。我们在计算机屏幕上看到的圆形并不是由完美的连续的线组成的；而是用一个个连续的像素点来画出的一个近似圆。地理形状的工作方式就与此相似。

复杂的形状 -- 比如 点集，线，多边形，多多变形，中空多边形等 -- 都是通过一个个 **geohash** 单元来画出的。这些形状会转化为一个被它所覆盖到的 **geohash** 集合。

注意

实际上，有两种类型的格子模式能用于地理星座：默认是使用我们之前讨论过的 **geohash**；另外还有一种是象限4叉树 (**quad trees**)。象限4叉树和**geohash**类似，只不过它每个层级都是4个单元（而不是像**geohash**一样的32个）。这种不同取决于编码方式的选择。

组成一个形状的 **geohash** 都作为一个组索引在一起。有这些信息，通过查看是否有相同的 **geohash** 单元，就可以很轻易地检查两个形状是否有交集。

地理形状有这些用处：判断查询的形状与索引的形状的关系；这些关系可能是以下之一：

- `intersects ::`

查询的形状与索引形状有重叠（默认）。

- `disjoint ::`

查询的形状与索引的形状完全不重叠。

- `within ::`

索引的形状完全被包含在查询形状中。

注意

地理形状不能用语计算距离、排序、打分以及聚集。



映射地理形状

与 `geo_point` 类型的字段相似，地理形状也需要在使用前明确映射：

```
PUT /attractions
{
  "mappings": {
    "landmark": {
      "properties": {
        "name": {
          "type": "string"
        },
        "location": {
          "type": "geo_shape"
        }
      }
    }
  }
}
```

你需要关注两个重要的设置项来调整精度（`precision`）和距离误差

（`distance_error_pct`）。There are two important settings that you should consider changing `precision` and `distance_error_pct`.

精度

精度（`precision`）参数用来控制组成地理形状的geohash的长度。它的默认值是 `9`，等同于尺寸在 `5m*5m` 的geohash。这个精度可能比你需要的精确得多。

精度越低，需要索引的单元就越少，检索时也会更快。当然，精度越低，地理形状的准确性就越差。你需要决定自己的地理形状所需精度——即使减少1-2个等级的精度也能带来明显的消耗缩减收益。

你可以通过指定距离的方式来定制精度——比如，`50m` 或 `2km`；不过这些距离最终也是会转换成对应的geohash长度（见 [geohashes](#)）。

距离误差

当索引一个多边形时，中间连续区域很容易用一个短geohash来表示。麻烦的是边缘部分，这些地方需要使用更精细的geohash才能表示。



当你在索引一个小地标时，你希望它的边界比较精确；（如果精度不够，）让这些纪念碑一个叠着一个可不好。当索引整个国家时，你就不需要这么高的精度。误差个50米左右也没什么大不了。

距离误差（`distance_error_pct`）指定地理形状可以接受的最大错误率。它的默认值是`0.025`，即`2.5%`。这也就是说，大的地理形状（比如国家）相比小的地理形状（比如纪念碑）来说，容许更加模糊的边界。

`0.025` 是一个不错的初始值。不过如果我们容许更大的错误率，对应地理形状需要索引的单元就越少。



索引地理形状

地理形状通过GeoJSON来表示，这是一种开放的使用JSON实现的二维形状编码方式。每个形状包含两个信息：形状类型：`point`，`line`，`polygon`，`envelope`；一个或多经度点集合的数组。

注意：

在GeoJSON里，经度表示方式通常是“纬度在前，经度在后”。

举例如下，我们用一个多边形来索引阿姆斯特丹达姆广场：

```
PUT /attractions/landmark/dam_square
{
  "name" : "Dam Square, Amsterdam",
  "location" : {
    "type" : "polygon", <1>
    "coordinates" : [[ <2>
      [ 4.89218, 52.37356 ],
      [ 4.89205, 52.37276 ],
      [ 4.89301, 52.37274 ],
      [ 4.89392, 52.37250 ],
      [ 4.89431, 52.37287 ],
      [ 4.89331, 52.37346 ],
      [ 4.89305, 52.37326 ],
      [ 4.89218, 52.37356 ]
    ]]
  }
}
```

- <1> `type` 参数指明如何使用经度坐标集来表示对应形状。
- <2> 用来表示多边形的经度坐标点列表。

上例中大量的方括号可能看起来让人困惑，不过实际上GeoJSON的语法非常简单：

1. 用一个数组表示经度坐标点：

```
[lon, lat]
```

2. 一组坐标点放到一个数组来表示一个多边形：

```
[[lon, lat], [lon, lat], ... ]
```



3. 一个[多边形](#)（`polygon`）形状可以包含多个多边形；第一个表示多边形的外轮廓，后续的多边形表示第一个多边形内部的空洞：

```
[  
  [[lon,lat],[lon,lat], ... ], # main polygon  
  [[lon,lat],[lon,lat], ... ], # hole in main polygon  
  ...  
]
```

参见 [Geo-shape mapping documentation](#) 了解更多支持的形状。



查询地理形状

地理形状一个不寻常的地方在于它运行我们使用形状来做查询，而不仅仅是坐标点。

举个例子，当我们的用户刚刚迈出阿姆斯特丹中央火车站时，我们可以用如下方式，查询出方圆1km内所有的地标：

```
GET /attractions/landmark/_search
{
  "query": {
    "geo_shape": {
      "location": { <1>
        "shape": { <2>
          "type": "circle", <3>
          "radius": "1km"
          "coordinates": [ <4>
            4.89994,
            52.37815
          ]
        }
      }
    }
  }
}
```

- <1> 查询使用 `location` 字段中的地理形状；
- <2> 查询中的形状是由 `shape` 键对应的内容表示；
- <3> 形状是一个半径为1km的圆形；
- <4> 安姆斯特丹中央火车站入口的坐标点。

默认，查询（或者过滤器——工作方式相同）会从已索引的形状中寻找与指定形状有交集的形状。此外，`relation` 也可以设置为 `disjoint` 来查找与指定形状不相交的，或者设置为 `within` 来查找完全落在查询形状中的。

举个例子，我们查找所有落在阿姆斯特丹内的地标：



```
GET /attractions/landmark/_search
{
  "query": {
    "geo_shape": {
      "location": {
        "relation": "within", <1>
        "shape": {
          "type": "polygon",
          "coordinates": [[ <2>
            [4.88330,52.38617],
            [4.87463,52.37254],
            [4.87875,52.36369],
            [4.88939,52.35850],
            [4.89840,52.35755],
            [4.91909,52.36217],
            [4.92656,52.36594],
            [4.93368,52.36615],
            [4.93342,52.37275],
            [4.92690,52.37632],
            [4.88330,52.38617]
          ]]
        }
      }
    }
  }
}
```

- <1> 只匹配完全落在查询形状中的（已索引）形状。
- <2> 这个多边形表示安姆斯特丹中心。



在查询中使用已索引的形状

对于那些经常会在查询中使用的形状，可以把它们索引起来以便在查询中可以方便地直接引用名字。以之前的阿姆斯特丹中央为例，我们可以把它存储为一个类型为 `neighborhood` 的文档。

首先，我们仿照之前设置 `landmark` 时的方式建立一个映射：

```
PUT /attractions/_mapping/neighborhood
{
  "properties": {
    "name": {
      "type": "string"
    },
    "location": {
      "type": "geo_shape"
    }
  }
}
```

然后我们索引阿姆斯特丹中央对应的形状：

```
PUT /attractions/neighborhood/central_amsterdam
{
  "name" : "Central Amsterdam",
  "location" : {
    "type" : "polygon",
    "coordinates" : [
      [4.88330, 52.38617],
      [4.87463, 52.37254],
      [4.87875, 52.36369],
      [4.88939, 52.35850],
      [4.89840, 52.35755],
      [4.91909, 52.36217],
      [4.92656, 52.36594],
      [4.93368, 52.36615],
      [4.93342, 52.37275],
      [4.92690, 52.37632],
      [4.88330, 52.38617]
    ]
  }
}
```

形状索引好之后，我们就可以在查询中通过 `index`、`type` 和 `id` 来引用它了：



```
GET /attractions/landmark/_search
{
  "query": {
    "geo_shape": {
      "location": {
        "relation": "within",
        "indexed_shape": { <1>
          "index": "attractions",
          "type": "neighborhood",
          "id": "central_amsterdam",
          "path": "location"
        }
      }
    }
  }
}
```

<1> 指定 `indexed_shape` 而不是 `shape`，Elasticsearch 就知道需要从指定的文档和路径检索出对应的形状了。

阿姆斯特丹中央这个形状没有什么特别的。同样地，我们也可以使用已经索引好的阿姆斯特丹达姆广场。这个查询查找出与阿姆斯特丹达姆广场有交集的临近点：

```
GET /attractions/neighborhood/_search
{
  "query": {
    "geo_shape": {
      "location": {
        "indexed_shape": {
          "index": "attractions",
          "type": "landmark",
          "id": "dam_square",
          "path": "location"
        }
      }
    }
  }
}
```



地理形状的过滤与缓存

地理形状 的查询和过滤都是表现为相同功能。查询就是简单的表现为一个过滤：把所以匹配到的文档的 `_score` 标记为1。 查询结果不能被缓存，不过过滤结果可以。

结果默认是不被缓存的。与地理坐标点集类似，任何形状内坐标的变化都会导致 geohash 集合的变化，因此在缓存过滤结果几乎没有意义。也就是说，除非你会重复的使用相同的形状来做过滤，它才是值得缓存起来的。 缓存方法是，把 `_cache` 设置为 `true`：

```
GET /attractions/neighborhood/_search
{
  "query": {
    "filtered": {
      "filter": {
        "geo_shape": {
          "_cache": true, <1>
          "location": {
            "indexed_shape": {
              "index": "attractions",
              "type": "landmark",
              "id": "dam_square",
              "path": "location"
            }
          }
        }
      }
    }
  }
}
```

- <1> `geo_shape` 过滤器的结果将被缓存。



[[relations]] == Handling Relationships

In the real world, relationships(("relationships")) matter: blog posts have comments, bank accounts have transactions, customers have bank accounts, orders have order lines, and directories have files and subdirectories.

Relational databases are specifically designed--and this will not come as a surprise to you--to manage(("relational databases", "managing relationships")) relationships:

- Each entity (or *row*, in the relational world) can be uniquely identified by a *primary key*. ((("primary key")))
- Entities are *normalized*. The data for a unique entity is stored only once, and related entities store just its primary key. Changing the data of an entity has to happen in only one place.((("joins", "in relational databases")))
- Entities can be joined at query time, allowing for cross-entity search.
- Changes to a single entity are *atomic*, *consistent*, *isolated*, and *durable*. (See [http://en.wikipedia.org/wiki/ACID_transactions\[ACID Transactions\]](http://en.wikipedia.org/wiki/ACID_transactions[ACID Transactions]) for more on this subject.)
- Most relational databases support ACID transactions across multiple entities.

But relational ((("ACID transactions"))) databases do have their limitations, besides their poor support for full-text search. Joining entities at query time is expensive--more joins that are required, the more expensive the query. Performing joins between entities that live on different hardware is so expensive that it is just not practical. This places a limit on the amount of data that can be stored on a single server.

Elasticsearch, like(("NoSQL databases")) most NoSQL databases, treats the world as though it were flat. An index is a flat collection of independent documents.((("indices")))) A single document should contain all of the information that is required to decide whether it matches a search request.

While changing the data of a single document in Elasticsearch is [http://en.wikipedia.org/wiki/ACID_transactions\[ACIDic\]](http://en.wikipedia.org/wiki/ACID_transactions[ACIDic]), transactions involving multiple documents are not. There is no way to roll back the index to its previous state if part of a transaction fails.

This FlatWorld has its advantages:

- Indexing is fast and lock-free.
- Searching is fast and lock-free.
- Massive amounts of data can be spread across multiple nodes, because each document is independent of the others.



But relationships matter. Somehow, we need to bridge the gap between FlatWorld and the real world.((("relationships", "techniques for managing relational data in Elasticsearch")))
Four common techniques are used to manage relational data in Elasticsearch:

- <>
- <>
- <>
- <>

Often the final solution will require a mixture of a few of these techniques.



应用级别的Join操作

我们可以在应用这一层面（部分的）模仿实现关系数据库中的join操作。例如，我们要给 `users` 以及每个 `user` 所对应的若干篇 `blog` 建立索引。在这充满关系的世界中，我们可以做一些类似于这样的事情：

```
PUT /my_index/user/1      (1)
{
  "name":      "John Smith",
  "email":     "john@smith.com",
  "dob":       "1970/10/24"
}

PUT /my_index/blogpost/2 (1)
{
  "title":     "Relationships",
  "body":      "It's complicated...",
  "user":      1           (2)
}
```

(1) 每一个 document 中 `index`，`type`，和 `id` 共同组成了主键。

(2) `blogpost` 通过包含 `user` 的 `id` 来关联 `user`，而这里不需要指定 `user` 的 `index` 和 `type` 是因为在我们的应用中它们是被硬编码的（这里的硬编码的意思应该是说，在 `blogpost` document 中引用了 `user`，那么es就会在相同的index下查找 `user type`，并且id为1的document，所以不需要指定 `index` 和 `type`）。

通过查询 `user` 的ID为1将很容易找到相应的 `blog`：

```
GET /my_index/blogpost/_search
{
  "query": {
    "filtered": {
      "filter": {
        "term": { "user": 1 }
      }
    }
  }
}
```

想通过博客作者的名字 `John` 来找到相关的博客，我们需要执行2个查询语句：第一，我们需要先找到所有叫 `John` 的博客作者，从而获得它们的 ID列表，第二，将获取到的ID列表作为查询条件来执行类似于上面的查询语句：



```
GET /my_index/user/_search
{
  "query": {
    "match": {
      "name": "John"
    }
  }
}

GET /my_index/blogpost/_search
{
  "query": {
    "filtered": {
      "filter": {
        "terms": { "user": [1] } (1)
      }
    }
  }
}
```

(1) 其中 `terms` 的值被设置成从第一个查询中得到的ID列表。

在应用级别模仿join操作的最大好处是数据是立体的（normalized），如果想改变 `user` 的姓名，那么只要在 `user` 这个 `document` 上改就可以了。而缺点是你必须在查询期间运行额外的 `query` 来实现 `join` 的操作。

在这个例子当中，只有一个 `user` 符合我们的第一个查询条件，但在真实的世界中，很可能会出现数百万人的名字叫 `John`，将这么多的ID塞到第二个查询中，将会让这个查询语句变得非常庞大，并且这个查询会执行数百万次 `term` 的查找。

这种模仿join操作的方法适合于前置查询结果集（在该例子中指代 `user`）比较小，并且最好是不经常变化的，此时我们在应用中可以去缓存这部分数据，避免频繁的执行第一个查询。



扁平化你的数据

Elasticsearch 鼓励你在创建索引的时候就 **扁平化 (denormalizing)** 你的数据，这样做可以获取最好的搜索性能。在每一篇文档里面冗余一些数据可以避免join操作。

举个例子，如果我们想通过 `user` 来查找某一篇 `blog`，那么就把 `user` 的姓名包含在 `blog` 这个 `document` 里面，就像这样：

```
PUT /my_index/user/1
{
  "name": "John Smith",
  "email": "john@smith.com",
  "dob": "1970/10/24"
}

PUT /my_index/blogpost/2
{
  "title": "Relationships",
  "body": "It's complicated...",
  "user": {
    "id": 1,
    "name": "John Smith" (1)
  }
}
```

(1) `user` 中的一部分信息已经被包含到 `blogpost` 里面。

现在，我们只要通过一次查询，就能找到和作者 `John` 有关系的所有博客：

```
GET /my_index/blogpost/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title": "relationships" } },
        { "match": { "user.name": "John" } }
      ]
    }
  }
}
```

扁平化数据的好处就是一个字，快。因为每一篇文档都已经包含了所有需要被查询的信息，所以就没有必要去做消耗很大的join操作了。



[[top-hits]] === Field Collapsing

A common requirement is the need to present search results grouped by a particular field. (((("field collapsing"))))(((("relationships", "field collapsing"))))We might want to return the most relevant blog posts *grouped* by the user's name. (((("terms aggregation"))))(((("aggregations", "field collapsing")))) Grouping by name implies the need for a `terms` aggregation. To be able to group on the user's *whole* name, the name field should be available in its original `not_analyzed` form, as explained in <>:

[source,json]

```
PUT /my_index/_mapping/blogpost { "properties": { "user": { "properties": { "name": { <1>
"type": "string", "fields": { "raw": { <2> "type": "string", "index": "not_analyzed" } } } } } }
```

```
}
```

<1> The `user.name` field will be used for full-text search.

<2> The `user.name.raw` field will be used for grouping with the `terms` aggregation.

Then add some data:

[source,json]

```
PUT /my_index/user/1 { "name": "John Smith", "email": "john@smith.com", "dob":
"1970/10/24" }
```

```
PUT /my_index/blogpost/2 { "title": "Relationships", "body": "It's complicated...", "user": { "id": 1, "name": "John Smith" } }
```

```
PUT /my_index/user/3 { "name": "Alice John", "email": "alice@john.com", "dob": "1979/01/04" }
```

```
PUT /my_index/blogpost/4 { "title": "Relationships are cool", "body": "It's not complicated at all...", "user": { "id": 3, "name": "Alice John" } }
```

```
}
```

Now we can run a query looking for blog posts about `relationships`, by users called `John`, and group the results by user, thanks to the <http://bit.ly/1CrlWFQ>[``top_hits`` aggregation]:



[source,json]

```
GET /my_index/blogpost/_search?search_type=count <1> { "query": { <2> "bool": { "must": [ { "match": { "title": "relationships" }}, { "match": { "user.name": "John" }} ] } }, "aggs": { "users": { "terms": { "field": "user.name.raw", <3> "order": { "top_score": "desc" } <4> }, "aggs": { "top_score": { "max": { "script": "_score" }}, <4> "blogposts": { "top_hits": { "_source": "title", "size": 5 } } <5> } } }
```

{}

<1> The blog posts that we are interested in are returned under the `blogposts` aggregation, so we can disable the usual search `hits` by setting the `search_type=count`.

<2> The `query` returns blog posts about `relationships` by users named `John`.

<3> The `terms` aggregation creates a bucket for each `user.name.raw` value.

<4> The `top_score` aggregation orders the terms in the `users` aggregation by the top-scoring document in each bucket.

<5> The `top_hits` aggregation returns just the `title` field of the five most relevant blog posts for each user.

The abbreviated response is shown here:

[source,json]

```
... "hits": { "total": 2, "max_score": 0, "hits": [] <1> }, "aggregations": { "users": { "buckets": [ { "key": "John Smith", <2> "doc_count": 1, "blogposts": { "hits": { <3> "total": 1, "max_score": 0.35258877, "hits": [ { "_index": "my_index", "_type": "blogpost", "_id": "2", "_score": 0.35258877, "_source": { "title": "Relationships" } } ] } }, "top_score": { <4> "value": 0.3525887727737427 } } },
```

...

<1> The `hits` array is empty because we set `search_type=count`.

<2> There is a bucket for each user who appeared in the top results.

<3> Under each user bucket there is a `blogposts.hits` array containing the top results for that user.



<4> The user buckets are sorted by the user's most relevant blog post.

Using the `top_hits` aggregation is the(((`"top_hits aggregation"`))) equivalent of running a query to return the names of the users with the most relevant blog posts, and then running the same query for each user, to get their best blog posts. But it is much more efficient.

The top hits returned in each bucket are the result of running a light *mini-query* based on the original main query. The mini-query supports the usual features that you would expect from search such as highlighting and pagination.



[[denormalization-concurrency]] === Denormalization and Concurrency

Of course, data denormalization has downsides too.(((“relationships”, “denormalization and concurrency”)))(((“concurrency”, “denormalization and”)))(((“denormalization”, “and concurrency”))) The first disadvantage is that the index will be bigger because the `_source` document for every blog post is bigger, and there are more indexed fields. This usually isn't a huge problem. The data written to disk is highly compressed, and disk space is cheap. Elasticsearch can happily cope with the extra data.

The more important issue is that, if the user were to change his name, all of his blog posts would need to be updated too. Fortunately, users don't often change names. Even if they did, it is unlikely that a user would have written more than a few thousand blog posts, so updating blog posts with the `<>` and `<>` APIs would take less than a second.

However, let's consider a more complex scenario in which changes are common, far reaching, and, most important, concurrent.(((“files”, “searching for files in a particular directory”)))

In this example, we are going to emulate a filesystem with directory trees in Elasticsearch, much like a filesystem on Linux: the root of the directory is `/`, and each directory can contain files and subdirectories.

We want to be able to search for files that live in a particular directory, the equivalent of this:

```
grep "some text" /clinton/projects/elasticsearch/*
```

This requires us to index the path of the directory where the file lives:

[source,json]

```
PUT /fs/file/1 { "name": "README.txt", <1> "path": "/clinton/projects/elasticsearch", <2> "contents": "Starting a new Elasticsearch project is easy..."}
```

```
}
```

<1> The filename

<2> The full path to the directory holding the file

[NOTE]



Really, we should also index `directory` documents so we can list all files and subdirectories within a directory, but for brevity's sake, we will ignore that requirement.

=====

We also want to be able to search for files that live anywhere in the directory tree below a particular directory, the equivalent of this:

```
grep -r "some text" /clinton
```

To support this, we need to index the path hierarchy:

- `/clinton`
- `/clinton/projects`
- `/clinton/projects/elasticsearch`

This hierarchy can be generated (((`"path_hierarchy tokenizer"`))) automatically from the `path` field using the [http://bit.ly/1AjGltZ``path_hierarchy`` tokenizer](http://bit.ly/1AjGltZ):

[source,json]

```
PUT /fs { "settings": { "analysis": { "analyzer": { "paths": { <1> "tokenizer": "path_hierarchy" } } } }
```

```
}
```

<1> The custom `paths` analyzer uses the `path_hierarchy` tokenizer with its default settings. See [http://bit.ly/1AjGltZ``path_hierarchy`` tokenizer](http://bit.ly/1AjGltZ`<code>path_hierarchy</code>` tokenizer).

The mapping for the `file` type would look like this:

[source,json]

```
PUT /fs/_mapping/file { "properties": { "name": { <1> "type": "string", "index": "not_analyzed" }, "path": { <2> "type": "string", "index": "not_analyzed", "fields": { "tree": { <2> "type": "string", "analyzer": "paths" } } } }
```

```
}
```

<1> The `name` field will contain the exact name.



<2> The `path` field will contain the exact directory name, while the `path.tree` field will contain the path hierarchy.

Once the index is set up and the files have been indexed, we can perform a search for files containing `elasticsearch` in just the `/clinton/projects/elasticsearch` directory like this:

[source,json]

```
GET /fs/file/_search { "query": { "filtered": { "query": { "match": { "contents": "elasticsearch" } }, "filter": { "term": { <1> "path": "/clinton/projects/elasticsearch" } } } }
```

```
}
```

<1> Find files in this directory only.

Every file that lives in any subdirectory under `/clinton` will include the term `/clinton` in the `path.tree` field. So we can search for all files in any subdirectory of `/clinton` as follows:

[source,json]

```
GET /fs/file/_search { "query": { "filtered": { "query": { "match": { "contents": "elasticsearch" } }, "filter": { "term": { <1> "path.tree": "/clinton" } } } }
```

```
}
```

<1> Find files in this directory or in any of its subdirectories.

==== Renaming Files and Directories

So far, so good.((("optimistic concurrency control")))((("files", "renaming files and directories"))) Renaming a file is easy--a simple `update` or `index` request is all that is required. You can even use `<>` to ensure that your change doesn't conflict with the changes from another user:

[source,json]

```
PUT /fs/file/1?version=2 <1> { "name": "README.asciidoc", "path": "/clinton/projects/elasticsearch", "contents": "Starting a new Elasticsearch project is easy..." }
```



<1> The `version` number ensures that the change is applied only if the document in the index has this same version number.

We can even rename a directory, but this means updating all of the files that exist anywhere in the path hierarchy beneath that directory. This may be quick or slow, depending on how many files need to be updated. All we would need to do is to use <> to retrieve all the files, and the <> to update them. The process isn't atomic, but all files will quickly move to their new home.



[[Concurrency-solutions]] === Solving Concurrency Issues

The problem comes when we want to allow more than one person to rename files or directories *at the same time*. (((“concurrency”, “solving concurrency issues”))) (((“relationships”, “solving concurrency issues”))) Imagine that you rename the `/clinton` directory, which contains hundreds of thousands of files. Meanwhile, another user renames the single file `/clinton/projects/elasticsearch/README.txt`. That user’s change, although it started after yours, will probably finish more quickly.

One of two things will happen:

- You have decided to use `version` numbers, in which case your mass rename will fail with a version conflict when it hits the renamed `README.asciidoc` file.
- You didn’t use versioning, and your changes will overwrite the changes from the other user.

The problem is that Elasticsearch does not support

http://en.wikipedia.org/wiki/ACID_transactions[ACID transactions].((“ACID transactions”)))

Changes to individual documents are ACIDic, but not changes involving multiple documents.

If your main data store is a relational database, and Elasticsearch is simply being used as a search engine(((“relational databases”, “Elasticsearch used with”))) or as a way to improve performance, make your changes in the database first and replicate those changes to Elasticsearch after they have succeeded. This way, you benefit from the ACID transactions available in the database, and all changes to Elasticsearch happen in the right order. Concurrency is dealt with in the relational database.

If you are not using a relational store, these concurrency issues need to be dealt with at the Elasticsearch level. The following are three practical solutions using Elasticsearch, all of which involve some form of locking:

- Global Locking
- Document Locking
- Tree Locking

[TIP]

The solutions described in this section could also be implemented by applying the same principles while using an external system instead of Elasticsearch.

=====

[[global-lock]] === Global Locking



We can avoid concurrency issues completely by allowing only one process to make changes at any time.((("locking", "global lock")))(("global lock")) Most changes will involve only a few files and will complete very quickly. A rename of a top-level directory may block all other changes for longer, but these are likely to be much less frequent.

Because document-level changes in Elasticsearch are ACIDic, we can use the existence or absence of a document as a global lock. To request a lock, we try to `create` the global-lock document:

[source,json]

```
PUT /fs/lock/global/_create
```

```
{}
```

If this `create` request fails with a conflict exception, another process has already been granted the global lock and we will have to try again later. If it succeeds, we are now the proud owners of the global lock and we can continue with our changes. Once we are finished, we must release the lock by deleting the global lock document:

[source,json]

DELETE /fs/lock/global

Depending on how frequent changes are, and how long they take, a global lock could restrict the performance of a system significantly. We can increase parallelism by making our locking more fine-grained.

[[document-locking]] ===== Document Locking

Instead of locking the whole filesystem, we could lock individual documents by using the same technique as previously described.((("locking", "document locking")))(("document locking")) A process could use a `<>` request to retrieve the IDs of all documents that would be affected by the change, and would need to create a lock file for each of them:

[source,json]



```
PUT /fs/lock/_bulk { "create": { "_id": 1 } } <1> { "process_id": 123 } <2> { "create": { "_id": 2 } } { "process_id": 123 }
```

...

<1> The ID of the `lock` document would be the same as the ID of the file that should be locked.

<2> The `process_id` is a unique ID that represents the process that wants to perform the changes.

If some files are already locked, parts of the `bulk` request will fail and we will have to try again.

Of course, if we try to lock *all* of the files again, the `create` statements that we used previously will fail for any file that is already locked by us! Instead of a simple `create` statement, we need an `update` request with an `upsert` parameter and this `script`:

[source,groovy]

```
if ( ctx._source.process_id != process_id ) { <1> assert false; <2> }
```

ctx.op = 'noop'; <3>

<1> `process_id` is a parameter that we pass into the script.

<2> `assert false` will throw an exception, causing the update to fail.

<3> Changing the `op` from `update` to `noop` prevents the update request from making any changes, but still returns success.

The full `update` request looks like this:

[source,json]

```
POST /fs/lock/1/_update { "upsert": { "process_id": 123 }, "script": "if ( ctx._source.process_id != process_id ) { assert false }; ctx.op = 'noop';" "params": { "process_id": 123 } }
```

}



If the document doesn't already exist, the `upsert` document will be inserted--much the same as the `create` request we used previously. However, if the document *does* exist, the script will look at the `process_id` stored in the document. If it is the same as ours, it aborts the update (`noop`) and returns success. If it is different, the `assert false` throws an exception and we know that the lock has failed.

Once all locks have been successfully created, the rename operation can begin. Afterward, we must release((("delete-by-query request"))) all of the locks, which we can do with a `delete-by-query` request:

[source,json]

```
POST /fs/_refresh <1>  
DELETE /fs/lock/_query { "query": { "term": { "process_id": 123 } } }
```

```
}
```

<1> The `refresh` call ensures that all `lock` documents are visible to the `delete-by-query` request.

Document-level locking enables fine-grained access control, but creating lock files for millions of documents can be expensive. In certain scenarios, such as this example with directory trees, it is possible to achieve fine-grained locking with much less work.

[[tree-locking]] ===== Tree Locking

Rather than locking every involved document, as in the previous option, we could lock just part of the directory tree.((("locking", "tree locking"))) We will need exclusive access to the file or directory that we want to rename, which can be achieved with an *exclusive lock* document:

[source,json]

```
{ "lock_type": "exclusive" }
```

And we need shared locks on any parent directories, with a *shared lock* document:

[source,json]



```
{ "lock_type": "shared", "lock_count": 1 <1>
```

```
}
```

<1> The `lock_count` records the number of processes that hold a shared lock.

A process that wants to rename `/clinton/projects/elasticsearch/README.txt` needs an *exclusive* lock on that file, and a *shared* lock on `/clinton`, `/clinton/projects`, and `/clinton/projects/elasticsearch`.

A simple `create` request will suffice for the exclusive lock, but the shared lock needs a scripted update to implement some extra logic:

[source,groovy]

```
if (ctx._source.lock_type == 'exclusive') { assert false; <1> }
```

ctx._source.lock_count++ <2>

<1> If the `lock_type` is `exclusive`, the `assert` statement will throw an exception, causing the update request to fail.

<2> Otherwise, we increment the `lock_count`.

This script handles the case where the `lock` document already exists, but we will also need an `upsert` document to handle the case where it doesn't exist yet. The full update request is as follows:

[source,json]

```
POST /fs/lock/%2Fclinton/_update <1> { "upsert": { <2> "lock_type": "shared", "lock_count": 1 }, "script": "if (ctx._source.lock_type == 'exclusive') { assert false }; ctx._source.lock_count++"
```

```
}
```

<1> The ID of the document is `/clinton`, which is URL-encoded to `%2fclinton`.

<2> The `upsert` document will be inserted if the document does not already exist.



Once we succeed in gaining a shared lock on all of the parent directories, we try to `create` an exclusive lock on the file itself:

[source,json]

```
PUT /fs/lock/%2Fclinton%2fprojects%2felasticsearch%2fREADME.txt/_create
```

```
{ "lock_type": "exclusive" }
```

Now, if somebody else wants to rename the `/clinton` directory, they would have to gain an exclusive lock on that path:

[source,json]

```
PUT /fs/lock/%2Fclinton/_create
```

```
{ "lock_type": "exclusive" }
```

This request would fail because a `lock` document with the same ID already exists. The other user would have to wait until our operation is done and we have released our locks. The exclusive lock can just be deleted:

[source,json]

DELETE

```
/fs/lock/%2Fclinton%2fprojects%2felasticsearch%2fREADME.txt
```

The shared locks need another script that decrements the `lock_count` and, if the count drops to zero, deletes the `lock` document:

[source,groovy]

```
if (--ctx._source.lock_count == 0) { ctx.op = 'delete' <1>
```



<1> Once the `lock_count` reaches `0`, the `ctx.op` is changed from `update` to `delete`.

This update request would need to be run for each parent directory in reverse order, from longest to shortest:

[source,json]

```
POST /fs/lock/%2Fclinton%2fprojects%2felasticsearch/_update { "script": "if (--  
ctx._source.lock_count == 0) { ctx.op = 'delete' } "
```

}

Tree locking gives us fine-grained concurrency control with the minimum of effort. Of course, it is not applicable to every situation--the data model must have some sort of access path like the directory tree for it to work.

[NOTE]

None of the three options--global, document, or tree locking--deals with the thorniest problem associated with locking: what happens if the process holding the lock dies?

The unexpected death of a process leaves us with two problems:

- How do we know that we can release the locks held by the dead process?
- How do we clean up the change that the dead process did not manage to complete?

These topics are beyond the scope of this book, but you will need to give them some thought if you decide to use locking.

=====

While denormalization is a good choice for many projects, the need for locking schemes can make for complicated implementations. Instead, Elasticsearch provides two models that help us deal with related entities: *nested objects* and *parent-child relationships*.





嵌套-对象

嵌套对象

事实上在 Elasticsearch 中，创建、删除、修改一个文档是原子性的，因此我们可以在一个文档中储存密切关联的实体。举例来说，我们可以在一个文档中储存一笔订单及其所有内容，或是储存一个 Blog 文章及其所有回应，藉由传递一个 comments 阵列：

```
PUT /my_index/blogpost/1
{
  "title": "Nest eggs",
  "body": "Making your money work...",
  "tags": [ "cash", "shares" ],
  "comments": [ <1>
    {
      "name": "John Smith",
      "comment": "Great article",
      "age": 28,
      "stars": 4,
      "date": "2014-09-01"
    },
    {
      "name": "Alice White",
      "comment": "More like this please",
      "age": 31,
      "stars": 5,
      "date": "2014-10-22"
    }
  ]
}
```

<1> 如果我们依靠动态映射， comments 标位会被自动建立为一个 object 标位。

因为所有内容都在同一个文档中，使搜寻时并不需要连接(join)blog 文章与回应，因此搜寻表现更加优异。

问题在於以上的文档可能会如下所示的匹配一个搜寻：



```
GET /_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "name": "Alice" }},
        { "match": { "age": 28 }} <1>
      ]
    }
  }
}
```

<1> Alice是31岁，而不是28岁！

造成跨对象配对的原因如同我们在对象阵列中所讨论到，在于我们优美结构的JSON文档在索引中被扁平化为下方的键-值形式：

```
{
  "title": [ eggs, nest ],
  "body": [ making, money, work, your ],
  "tags": [ cash, shares ],
  "comments.name": [ alice, john, smith, white ],
  "comments.comment": [ article, great, like, more, please, this ],
  "comments.age": [ 28, 31 ],
  "comments.stars": [ 4, 5 ],
  "comments.date": [ 2014-09-01, 2014-10-22 ]
}
```

Alice 与 31 以及 John 与 2014-09-01 之间的关联已经无法挽回的消失了。当 object 类型的栏位用于储存单一对象是非常有用的。从搜寻的角度来看，对于排序一个对象阵列来说关联是不需要的东西。

这是嵌套对象被设计来解决的问题。藉由映射 comments 栏位为 nested 类型而不是 object 类型，每个嵌套对象会被索引为一个隐藏分割文档，例如：



```
{ <1>
  "comments.name": [ john, smith ],
  "comments.comment": [ article, great ],
  "comments.age": [ 28 ],
  "comments.stars": [ 4 ],
  "comments.date": [ 2014-09-01 ]
}
{ <2>
  "comments.name": [ alice, white ],
  "comments.comment": [ like, more, please, this ],
  "comments.age": [ 31 ],
  "comments.stars": [ 5 ],
  "comments.date": [ 2014-10-22 ]
}
{ <3>
  "title": [ eggs, nest ],
  "body": [ making, money, work, your ],
  "tags": [ cash, shares ]
}
```

<1> 第一个 嵌套 对象

<2> 第二个 嵌套 对象

<3> 根或是父文档

藉由分别索引每个嵌套对象，对象的栏位中保持了其关联。我们的查询可以只在同一个嵌套对象都匹配时才回应。

不仅如此，因嵌套对象都被索引了，连接嵌套对象至根文档的查询速度非常快--几乎与查询单一文档一样快。

这些额外的嵌套对象被隐藏起来，我们无法直接访问他们。为了要新增、修改或移除一个嵌套对象，我们必须重新索引整个文档。要牢记搜寻要求的结果并不是只有嵌套对象，而是整个文档。



嵌套-映射

嵌套对象映射

设定一个 `nested` 样式很简单--在你会设定为 `object` 类型的地方，改为 `nested` 类型：

```
PUT /my_index
{
  "mappings": {
    "blogpost": {
      "properties": {
        "comments": {
          "type": "nested", <1>
          "properties": {
            "name": { "type": "string" },
            "comment": { "type": "string" },
            "age": { "type": "short" },
            "stars": { "type": "short" },
            "date": { "type": "date" }
          }
        }
      }
    }
  }
}
```

<1> 一个 `nested` 样式接受与 `object` 类型相同的参数。

所需仅此而已。任何 `comments` 对象会被索引为分离嵌套对象。参考更多 [nested type reference docs](#)。



嵌套-查询

查询嵌套对象

因嵌套对象(nested objects)会被索引为分离的隐藏文档，我们不能直接查询它们。而是使用 `nested` 查询或 `nested` 过滤器来存取它们：

```
GET /my_index/blogpost/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title": "eggs" }}, <1>
        {
          "nested": {
            "path": "comments", <2>
            "query": {
              "bool": {
                "must": [ <3>
                  { "match": { "comments.name": "john" }},
                  { "match": { "comments.age": 28     }}]
              }
            }}}
        ]
      }
    }
}
```

<1> `title` 条件运作在根文档上

<2> `nested` 条件 深入 嵌套的 `comments` 栏位。它不会在存取根文档的栏位，或是其他嵌套文档的栏位。

<3> `comments.name` 以及 `comments.age` 运作在相同的嵌套文档。

TIP

一个 `nested` 栏位可以包含其他 `nested` 栏位。相同的，一个 `nested` 查询可以包含其他 `nested` 查询。嵌套阶层会如同你预期的运作。

当然，一个 `nested` 查询可以匹配多个嵌套文档。每个文档的匹配会有各自的关联分数，但多个分数必须减少至单一分数才能应用至根文档。

在预设中，它会平均所有嵌套文档匹配的分数。这可以藉由设定 `score_mode` 参数为 `avg` , `max` , `sum` 或甚至 `none` (为了防止根文档永远获得 `1.0` 的匹配分数时)来控制。



```
GET /my_index/blogpost/_search
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title": "eggs" }},
        {
          "nested": {
            "path": "comments",
            "score_mode": "max", <1>
            "query": {
              "bool": {
                "must": [
                  { "match": { "comments.name": "john" }},
                  { "match": { "comments.age": 28 }}
                ]
              }
            }
          }
        ]
      }
    }
  }
}
```

<1> 从最匹配的嵌套文档中给予根文档的 `_score` 值。

注意

`nested` 过滤器类似於 `nested` 查询，除了无法使用 `score_mode` 参数。只能使用在 `filter context`—例如在 `filtered` 查询中--其作用类似其他的过滤器：包含或不包含，但不评分。

`nested` 过滤器的结果本身不会缓存，通常缓存规则会被应用於 `nested` 过滤器之中的过滤器。



嵌套排序

以嵌套栏位排序

我们可以依照嵌套栏位中的值来排序，甚至藉由分离嵌套文档中的值。为了使其结果更加有趣，我们加入另一个记录：

```
PUT /my_index/blogpost/2
{
  "title": "Investment secrets",
  "body": "What they don't tell you ...",
  "tags": [ "shares", "equities" ],
  "comments": [
    {
      "name": "Mary Brown",
      "comment": "Lies, lies, lies",
      "age": 42,
      "stars": 1,
      "date": "2014-10-18"
    },
    {
      "name": "John Smith",
      "comment": "You're making it up!",
      "age": 28,
      "stars": 2,
      "date": "2014-10-16"
    }
  ]
}
```

想像我们要取回在十月中有收到回应的blog文章，并依照所取回的各个blog文章中最少 stars 数量的顺序作排序。这个搜寻请求如下：



```
GET /_search
{
  "query": {
    "nested": { <1>
      "path": "comments",
      "filter": {
        "range": {
          "comments.date": {
            "gte": "2014-10-01",
            "lt": "2014-11-01"
          }
        }
      }
    },
    "sort": {
      "comments.stars": { <2>
        "order": "asc", <2>
        "mode": "min", <2>
        "nested_filter": { <3>
          "range": {
            "comments.date": {
              "gte": "2014-10-01",
              "lt": "2014-11-01"
            }
          }
        }
      }
    }
  }
}
```

<1> nested 查询限制了结果为十月份收到回应的blog文章。

<2> 结果在所有匹配的回应中依照 comment.stars 样位的最小值(min)作递增(asc)的排序。

<3> 排序条件中的 nested_filter 与主查询 query 条件中的 nested 查询相同。於下一个下方解释。

为什么我们要在 nested_filter 重复写上查询条件？原因是排序在於执行查询后才发生。此查询匹配了在十中有收到回应的blog文章，回传blog文章文档作为结果。如果我们不加上 nested_filter 条件，我们最後会依照任何blog文章曾经收到过的回应作排序，而不是在十月份收到的。



嵌套-集合

嵌套-集合

如同我们在查询时需要使用 `nested` 查询来存取嵌套对象，专门的 `nested` 集合使我们可以取得嵌套对象中栏位的集合：

```
GET /my_index/blogpost/_search?search_type=count
{
  "aggs": {
    "comments": {
      "nested": {
        "path": "comments"
      },
      "aggs": {
        "by_month": {
          "date_histogram": { <2>
            "field": "comments.date",
            "interval": "month",
            "format": "yyyy-MM"
          },
          "aggs": {
            "avg_stars": {
              "avg": { <3>
                "field": "comments.stars"
              }
            }
          }
        }
      }
    }
  }
}
```

<1> `nested` 集合 深入 嵌套对象的 `comments` 栏位

<2> 评论基於 `comments.date` 栏位被分至各个月份分段

<3> 每个月份分段单独计算星号的平均数

结果显示集合发生於嵌套文档层级：



```
...
"aggregations": {
  "comments": {
    "doc_count": 4, <1>
    "by_month": {
      "buckets": [
        {
          "key_as_string": "2014-09",
          "key": 1409529600000,
          "doc_count": 1, <1>
          "avg_stars": {
            "value": 4
          }
        },
        {
          "key_as_string": "2014-10",
          "key": 1412121600000,
          "doc_count": 3, <1>
          "avg_stars": {
            "value": 2.6666666666666665
          }
        }
      ]
    }
  }
}
...
<1> 此处总共有四个 comments :一个在九月以及三个在十月
```

反向-嵌套-集合

反向嵌套-集合

一个 `nested` 集合只能存取嵌套文档中的栏位，而无法看见根文档或其他嵌套文档中的栏位。然而，我们可以跳出嵌套区块，藉由 `reverse_nested` 集合回到父阶层。

举例来说，我们可以发现使用评论者的年龄为其加上 `tags` 很有趣。`comment.age` 是在嵌套栏位中，但是 `tags` 位於根文档：



```
GET /my_index/blogpost/_search?search_type=count
{
  "aggs": {
    "comments": {
      "nested": { <1>
        "path": "comments"
      },
      "aggs": {
        "age_group": {
          "histogram": { <2>
            "field": "comments.age",
            "interval": 10
          },
          "aggs": {
            "blogposts": {
              "reverse_nested": {}, <3>
              "aggs": {
                "tags": {
                  "terms": { <4>
                    "field": "tags"
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

<1> nested 集合深入 comments 对象

<2> histogram 集合以 comments.age 标位聚集成每十年一个的分段

<3> reverse_nested 集合跳回到根文档

<4> terms 集合计算每个年龄分段的火红词语

简略的结果显示如下：



```
...
"aggregations": {
  "comments": {
    "doc_count": 4, <1>
    "age_group": {
      "buckets": [
        {
          "key": 20, <2>
          "doc_count": 2, <2>
          "blogposts": {
            "doc_count": 2, <3>
            "tags": {
              "doc_count_error_upper_bound": 0,
              "buckets": [ <4>
                { "key": "shares", "doc_count": 2 },
                { "key": "cash", "doc_count": 1 },
                { "key": "equities", "doc_count": 1 }
              ]
            }
          }
        },
        ...
      ],
      ...
    }
  }
},
```

<1> 共有四个评论

<2> 有两个评论的发表者年龄介於20至30之间

<3> 两个blog文章与这些评论相关

<4> 这些blog文章的火红标签是 shares 、 cash 、 equities

什麼时候要使用嵌套对象

嵌套对象對於当有一个主要实体(如 blogpost)，加上有限数量的紧密相关实体(如 comments)是非常有用的。有办法能以评论内容找到blog文章很有用，且 nested 查询及过滤器提供短查询时间连接(fast query-time joins)。

嵌套模型的缺点如下：

- 如欲新增、修改或删除一个嵌套文档，则必须重新索引整个文档。因此越多嵌套文档造成越多的成本。
- 搜寻请求回传整个文档，而非只有匹配的嵌套文档。虽然有个进行中的计画要支持只回传根文档及最匹配的嵌套文档，但目前并未支持。

有时你需要完整分离主要文档及其关连实体。父-子关系提供这一个功能。





[[parent-child]] == Parent-Child Relationship

The *parent-child* relationship is (((("relationships", "parent-child")))((("parent-child relationship"))))similar in nature to the <>: both allow you to associate one entity with another. (((("nested objects", "parent-child relationships versus"))))The difference is that, with nested objects, all entities live within the same document while, with parent-child, the parent and children are completely separate documents.

The parent-child functionality allows you to associate one document type with another, in a *one-to-many* relationship--one parent to many children.(((("one-to-many relationships")))) The advantages that parent-child has over <> are as follows:

- The parent document can be updated without reindexing the children.
- Child documents can be added, changed, or deleted without affecting either the parent or other children. This is especially useful when child documents are large in number and need to be added or changed frequently.
- Child documents can be returned as the results of a search request.

Elasticsearch maintains a map of which parents are associated with which children. It is thanks to this map that query-time joins are fast, but it does place a limitation on the parent-child relationship: *the parent document and all of its children must live on the same shard*.

[NOTE]

At the time of going to press, the parent-child ID map is held in memory as part of <>. There are plans afoot to change the default setting to use <> by default instead.

=====

[[parent-child-mapping]] === Parent-Child Mapping

All that is needed in order to establish the parent-child relationship is to specify which document type should be the parent of a child type.(((("mapping (types)", "parent-child")))((("parent-child relationship", "parent-child mapping")))) This must be done at index creation time, or with the `update-mapping` API before the child type has been created.

As an example, let's say that we have a company that has branches in many cities. We would like to associate employees with the branch where they work. We need to be able to search for branches, individual employees, and employees who work for particular branches, so the nested model will not help. We could, of course, use <> or <> here instead, but for demonstration purposes we will use parent-child.



All that we have to do is to tell Elasticsearch that the `employee` type has the `branch` document type as its `_parent`, which we can do when we create the index:

[source,json]

```
PUT /company { "mappings": { "branch": {}, "employee": { "_parent": { "type": "branch" } } }}
```

```
}
```

<1> Documents of type `employee` are children of type `branch`.



[[indexing-parent-child]] === Indexing Parents and Children

Indexing parent documents is no different from any other document. Parents don't need to know anything about their children:

[source,json]

```
POST /company/branch/_bulk { "index": { "_id": "london" } } { "name": "London Westminster",  
"city": "London", "country": "UK" } { "index": { "_id": "liverpool" } } { "name": "Liverpool Central",  
"city": "Liverpool", "country": "UK" } { "index": { "_id": "paris" } }
```

```
{ "name": "Champs Élysées", "city": "Paris",  
"country": "France" }
```

When indexing child documents, you must specify the ID of the associated parent document:

[source,json]

```
PUT /company/employee/1?parent=london <1> { "name": "Alice Smith", "dob": "1970-10-  
24", "hobby": "hiking"
```

```
}
```

<1> This `employee` document is a child of the `london` branch.

This `parent` ID serves two purposes: it creates the link between the parent and the child, and it ensures that the child document is stored on the same shard as the parent.

In <>, we explained how Elasticsearch uses a routing value, which defaults to the `_id` of the document, to decide which shard a document should belong to. The routing value is plugged into this simple formula:

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

However, if a `parent` ID is specified, it is used as the routing value instead of the `_id`. In other words, both the parent and the child use the same routing value--the `_id` of the parent--and so they are both stored on the same shard.



The `parent` ID needs to be specified on all single-document requests: when retrieving a child document with a `GET` request, or when indexing, updating, or deleting a child document. Unlike a search request, which is forwarded to all shards in an index, these single-document requests are forwarded only to the shard that holds the document--if the `parent` ID is not specified, the request will probably be forwarded to the wrong shard.

The `parent` ID should also be specified when using the `bulk` API:

[source.json]

```
POST /company/employee/_bulk { "index": { "_id": 2, "parent": "london" }} { "name": "Mark Thomas", "dob": "1982-05-16", "hobby": "diving" } { "index": { "_id": 3, "parent": "liverpool" }} { "name": "Barry Smith", "dob": "1979-04-01", "hobby": "hiking" } { "index": { "_id": 4, "parent": "paris" }}
```

```
{ "name": "Adrien Grand", "dob": "1987-05-11", "hobby": "horses" }
```

WARNING: If you want to change the `parent` value of a child document, it is not sufficient to just reindex or update the child document--the new parent document may be on a different shard. Instead, you must first delete the old child, and then index the new child.



[[has-child]] === Finding Parents by Their Children

The `has_child` query and filter can be used to find parent documents based on the contents of their children.(((`has_child` query and filter)))((("parent-child relationship", "finding parents by their children"))) For instance, we could find all branches that have employees born after 1980 with a query like this:

[source,json]

```
GET /company/branch/_search { "query": { "has_child": { "type": "employee", "query": { "range": { "dob": { "gte": "1980-01-01" } } } } }
```

```
}
```

Like the `<>`, the `has_child` query could match several child documents,(((`has_child` query and filter", "query"))) each with a different relevance score. How these scores are reduced to a single score for the parent document depends on the `score_mode` parameter. The default setting is `none`, which ignores the child scores and assigns a score of `1.0` to the parents, but it also accepts `avg`, `min`, `max`, and `sum`.

The following query will return both `london` and `liverpool`, but `london` will get a better score because `Alice Smith` is a better match than `Barry Smith`:

[source,json]

```
GET /company/branch/_search { "query": { "has_child": { "type": "employee", "score_mode": "max", "query": { "match": { "name": "Alice Smith" } } } }
```

```
}
```

TIP: The default `score_mode` of `none` is significantly faster than the other modes because Elasticsearch doesn't need to calculate the score for each child document. Set it to `avg`, `min`, `max`, or `sum` only if you care about the score.((("parent-child relationship", "finding parents by their children", "min_children and max_children")))

[[min-max-children]] === min_children and max_children



The `has_child` query and filter both accept the `min_children` and `max_children` parameters, (((`"min_children parameter"`)))(((`"max_children parameter"`)))(((`"has_child query and filter", "min_children or max_children parameters"`))) which will return the parent document only if the number of matching children is within the specified range.

This query will match only branches that have at least two employees:

[source,json]

```
GET /company/branch/_search { "query": { "has_child": { "type": "employee", "min_children": 2, <1> "query": { "match_all": {} } } }}
```

```
}
```

<1> A branch must have at least two employees in order to match.

The performance of a `has_child` query or filter with the `min_children` or `max_children` parameters is much the same as a `has_child` query with scoring enabled.

.has_child Filter

The `has_child` filter works(((`"haschild query and filter"`, "filter"))) *in the same way as the `has_child` query, except that it doesn't support the `score_mode` parameter. It can be used only in `_filter` context—such as inside a `filtered` query—and behaves like any other filter: it includes or excludes, but doesn't score.*

While the results of a `has_child` filter are not cached, the usual caching rules apply to the filter *inside* the `has_child` filter.



[[has-parent]] === Finding Children by Their Parents

While a `nested` query can always (((“parent-child relationship”, “finding children by their parents”)))return only the root document as a result, parent and child documents are independent and each can be queried independently. The `has_child` query allows us to return parents based on data in their children, and the `has_parent` query returns children based on data in their parents.(((“has_parent query and filter”, “query”)))

It looks very similar to the `has_child` query. This example returns employees who work in the UK:

[source,json]

```
GET /company/employee/_search { "query": { "has_parent": { "type": "branch", <1> "query": { "match": { "country": "UK" } } } }
```

```
}
```

<1> Returns children who have parents of type `branch`

The `has_parent` query also supports the `score_mode`,(((“score_mode parameter”))) but it accepts only two settings: `none` (the default) and `score`. Each child can have only one parent, so there is no need to reduce multiple scores into a single score for the child. The choice is simply between using the `score` (`score`) or not (`none`).

.has_parent Filter

The `has_parent` filter works in the same way(((“hasparent query and filter”, “filter”))) as the `has_parent` query, except that it doesn’t support the `score_mode` parameter. It can be used only in `_filter` context—such as inside a `filtered` query—and behaves like any other filter: it includes or excludes, but doesn’t score.

While the results of a `has_parent` filter are not cached, the usual caching rules apply to the filter inside the `has_parent` filter.



[[children-agg]] === Children Aggregation

Parent-child supports a <http://bit.ly/1xtpjaz>[`children` aggregation] as (((("aggregations", "children aggregation"))))((("children aggregation")))))((("parent-child relationship", "children aggregation"))))a direct analog to the `nested` aggregation discussed in <>. A parent aggregation (the equivalent of `reverse_nested`) is not supported.

This example demonstrates how we could determine the favorite hobbies of our employees by country:

[source,json]

```
GET /company/branch/_search?search_type=count { "aggs": { "country": { "terms": { <1>  
"field": "country" }, "aggs": { "employees": { "children": { <2> "type": "employee" }, "aggs": {  
"hobby": { "terms": { <3> "field": "employee.hobby" } } } } } } }
```

```
}
```

<1> The `country` field in the `branch` documents.

<2> The `children` aggregation joins the parent documents with their associated children of type `employee` .

<3> The `hobby` field from the `employee` child documents.



[[grandparents]] === Grandparents and Grandchildren

The parent-child relationship can extend across more than one generation--grandchildren can (((("parent-child relationship", "grandparents and grandchildren"))))((("grandparents and grandchildren"))))have grandparents--but it requires an extra step to ensure that documents from all generations are indexed on the same shard.

Let's change our previous example to make the `country` type a parent of the `branch` type:

[source,json]

```
PUT /company { "mappings": { "country": {}, "branch": { "_parent": { "type": "country" <1> } }, "employee": { "_parent": { "type": "branch" <2> } } }
```

```
}
```

<1> `branch` is a child of `country`.

<2> `employee` is a child of `branch`.

Countries and branches have a simple parent-child relationship, so we use the same process as we used in <>:

[source,json]

```
POST /company/country/_bulk { "index": { "_id": "uk" } } { "name": "UK" } { "index": { "_id": "france" } } { "name": "France" }
```

```
POST /company/branch/_bulk { "index": { "_id": "london", "parent": "uk" } } { "name": "London Westminster" } { "index": { "_id": "liverpool", "parent": "uk" } } { "name": "Liverpool Central" } { "index": { "_id": "paris", "parent": "france" } }
```

{ "name": "Champs Élysées" }

The `parent` ID has ensured that each `branch` document is routed to the same shard as its parent `country` document. However, look what would happen if we were to use the same technique with the `employee` `grandchildren`:

[source,json]



```
PUT /company/employee/1?parent=london { "name": "Alice Smith", "dob": "1970-10-24",  
"hobby": "hiking"
```

{}

The shard routing of the employee document would be decided by the parent ID— `london` — but the `london` document was routed to a shard by *its own* parent ID— `uk` . It is very likely that the grandchild would end up on a different shard from its parent and grandparent, which would prevent the same-shard parent-child mapping from functioning.

Instead, we need to add an extra `routing` parameter, set to the ID of the grandparent, to ensure that all three generations are indexed on the same shard. The indexing request should look like this:

[source,json]

```
PUT /company/employee/1?parent=london&routing=uk <1> { "name": "Alice Smith", "dob":  
"1970-10-24", "hobby": "hiking"
```

{}

<1> The `routing` value overrides the `parent` value.

The `parent` parameter is still used to link the employee document with its parent, but the `routing` parameter ensures that it is stored on the same shard as its parent and grandparent. The `routing` value needs to be provided for all single-document requests.

Querying and aggregating across generations works, as long as you step through each generation. For instance, to find countries where employees enjoy hiking, we need to join countries with branches, and branches with employees:

[source,json]

```
GET /company/country/_search { "query": { "has_child": { "type": "branch", "query": {  
"has_child": { "type": "employee", "query": { "match": { "hobby": "hiking" } } } } } }
```

{}



[[parent-child-performance]] === Practical Considerations

Parent-child joins can be a useful technique for managing relationships when index-time performance(("parent-child relationship", "performance and")) is more important than search-time performance, but it comes at a significant cost. Parent-child queries can be 5 to 10 times slower than the equivalent nested query!

==== Memory Use

At the time of going to press, the parent-child ID map is still held in memory.(("parent-child relationship", "memory usage"))(("memory usage", "parent-child ID map")) There are plans to change the map to use doc values instead, which will be a big memory saving. Until that happens, you need to be aware of the following: the string `_id` field of every parent document has to be held in memory, and every child document requires 8 bytes (a long value) of memory. Actually, it's a bit less thanks to compression, but this gives you a rough idea.

You can check how much memory is being used by the parent-child cache by consulting ((("indices-stats API")))the `indices-stats` API (for a summary at the index level) or the `node-stats` API (for a summary at the node level):

[source,json]

GET /_nodes/stats/indices/id_cache?human <1>

<1> Returns memory use of the ID cache summarized by node in a human-friendly format.

==== Global Ordinals and Latency

Parent-child uses <> to speed(("global ordinals"))(("parent-child relationship", "global ordinals and latency")) up joins. Regardless of whether the parent-child map uses an in-memory cache or on-disk doc values, global ordinals still need to be rebuilt after any change to the index.

The more parents in a shard, the longer global ordinals will take to build. Parent-child is best suited to situations where there are many children for each parent, rather than many parents and few children.

Global ordinals, by default, are built lazily: the first parent-child query or aggregation after a refresh will trigger building of global ordinals. This can introduce a significant latency spike for your users. You can use <> to(("eager global ordinals")) shift the cost of building global



ordinals from query time to refresh time, by mapping the `_parent` field as follows:

[source,json]

```
PUT /company { "mappings": { "branch": {}, "employee": { "_parent": { "type": "branch", "fielddata": { "loading": "eager_global_ordinals" <1> } } } }
```

```
}
```

<1> Global ordinals for the `_parent` field will be built before a new segment becomes visible to search.

With many parents, global ordinals can take several seconds to build. In this case, it makes sense to increase the `refresh_interval` so(((`refresh_interval setting`))) that refreshes happen less often and global ordinals remain valid for longer. This will greatly reduce the CPU cost of rebuilding global ordinals every second.

==== Multigenerations and Concluding Thoughts

The ability to join multiple generations (see <>) sounds attractive until (((`grandparents` and `grandchildren`)))((("parent-child relationship", "multi-generations"))))you think of the costs involved:

- The more joins you have, the worse performance will be.
- Each generation of parents needs to have their string `_id` fields stored in memory, which can consume a lot of RAM.

As you consider your relationship schemes and whether parent-child is right for you, consider this advice (((`parent-child relationship`, "guidelines for using")))about parent-child relationships:

- Use parent-child relationships sparingly, and only when there are many more children than parents.
- Avoid using multiple parent-child joins in a single query.
- Avoid scoring by using the `has_child` filter, or the `has_child` query with `score_mode` set to `none`.
- Keep the parent IDs short, so that they require less memory.

Above all: think about the other relationship techniques that we have discussed before reaching for parent-child.



可扩展性规划

在某些公司Elasticsearch被用来每天索引和检索PB级别的数据，但大多情况我们是从一个相对小很多的数据集开始系统建设的。虽然我们希望能成为下一个Facebook，但是现实往往比理想更骨感。虽然我们从当下开始建设，但之后横向扩展的灵活性和简便性还是必须考虑的。

Elasticsearch天生就是可扩展的。无论是运行在你的个人电脑之上还是运行在由数百个节点构成的集群之上，Elasticsearch都能很好的工作而且两者之间的使用体验并不会有太大的差异。从一个小集群扩展到一个大集群的过程几乎是自动化的、自然而然的。从一个大集群扩展到一个巨型集群会需要一点规划与设计，但是相对来说还是比较自然的。

当然，Elasticsearch也不是包治百病的灵丹妙药，它也有自身的限制。如果你了解这些限制并且能很好的规避它们，这个扩展的过程将会比较平顺。否则你不善待Elasticsearch，它也不会让你很好过。

Elasticsearch的默认设置足以支撑你的系统走很长的路，但是为了更好的利用资源，你还是要认真地设计系统中的数据流。我们将会讨论两个通用的数据流场景：基于时间的数据流（比如日志事件、社交网络时间轴这些基于时间相关性的数据）和基于用户的数据流（比如大规模的数据集合可以按照用户/客户进行切分的场景）。

本章我们将帮助你在系统建设早期做出正确的决策，从而尽可能避免日后意想不到的麻烦。



[[shard-scale]] === The Unit of Scale

In <>, we explained that a shard is a *Lucene index* and that an Elasticsearch index is a collection of shards.((("scaling", "shard as unit of scale"))) Your application talks to an index, and Elasticsearch routes your requests to the appropriate shards.

A shard is the *unit of scale*. ((("shards", "as unit of scale"))) The smallest index you can have is one with a single shard. This may be more than sufficient for your needs--a single shard can hold a lot of data--but it limits your ability to scale.

Imagine that our cluster consists of one node, and in our cluster we have one index, which has only one shard:

[source,json]

```
PUT /my_index { "settings": { "number_of_shards": 1, <1> "number_of_replicas": 0 } }
```

```
}
```

<1> Create an index with one primary shard and zero replica shards.

This setup may be small, but it serves our current needs and is cheap to run.

[NOTE]

At the moment we are talking about only *primary* shards.((("primary shards"))) We discuss *replica* shards in <>.

=====

One glorious day, the Internet discovers us, and a single node just can't keep up with the traffic. We decide to add a second node, as per <>. What happens?

[[img-one-shard]] .An index with one shard has no scale factor

image::images/elas_4401.png["An index with one shard has no scale factor"]

The answer is: nothing. Because we have only one shard, there is nothing to put on the second node. We can't increase the number of shards in the index, because the number of shards is an important element in the algorithm used to <>:

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



Our only option now is to reindex our data into a new, bigger index that has more shards, but that will take time that we can ill afford. By planning ahead, we could have avoided this problem completely by *overallocating*.



[[overallocation]] === Shard Overallocation

A shard lives on a single node,((("scaling", "shard overallocation")))(("shards", "overallocation of"))) but a node can hold multiple shards. Imagine that we created our index with two primary shards instead of one:

[source,json]

```
PUT /my_index { "settings": { "number_of_shards": 2, <1> "number_of_replicas": 0 } }
```

```
}
```

<1> Create an index with two primary shards and zero replica shards.

With a single node, both shards would be assigned to the same node. From the point of view of our application, everything functions as it did before. The application communicates with the index, not the shards, and there is still only one index.

This time, when we add a second node, Elasticsearch will automatically move one shard from the first node to the second node, as depicted in <>. Once the relocation has finished, each shard will have access to twice the computing power that it had before.

[[img-two-shard]] .An index with two shards can take advantage of a second node
image::images/elas_4402.png["An index with two shards can take advantage of a second node"]

We have been able to double our capacity by simply copying a shard across the network to the new node. The best part is, we achieved this with zero downtime. All indexing and search requests continued to function normally while the shard was being moved.

A new index in Elasticsearch is allotted five primary shards by default. That means that we can spread that index out over a maximum of five nodes, with one shard on each node. That's a lot of capacity, and it happens without you having to think about it at all!

.Shard Splitting

Users often ask why Elasticsearch doesn't support *shard-splitting*—the ability to split each shard into two or more pieces. ((("shard splitting"))) The reason is that shard-splitting is a bad idea:



- Splitting a shard is almost equivalent to reindexing your data. It's a much heavier process than just copying a shard from one node to another.
- Splitting is exponential. You start with one shard, then split into two, and then four, eight, sixteen, and so on. Splitting doesn't allow you to increase capacity by just 50%.
- Shard splitting requires you to have enough capacity to hold a second copy of your index. Usually, by the time you realize that you need to scale out, you don't have enough free space left to perform the split.

In a way, Elasticsearch does support shard splitting. You can always reindex your data to a new index with the appropriate number of shards (see <>). It is still a more intensive process than moving shards around, and still requires enough free space to complete, but at least you can control the number of shards in the new index.



[[kagillion-shards]] === Kagillion Shards

The first thing that new users do when they learn about <> is((("scaling", "shard overallocation", "limiting")))((("shards", "overallocation of", "limiting")))) to say to themselves:

[quote, A new user]

[role="alignmeright"] I don't know how big this is going to be, and I can't change the index size later on, so to be on the safe side, I'll just give this index 1,000 shards...

One thousand shards--really? And you don't think that, perhaps, between now and the time you need to buy *one thousand nodes*, that you may need to rethink your data model once or twice and have to reindex?

A shard is not free. Remember:

- A shard is a Lucene index under the covers, which uses file handles, memory, and CPU cycles.
- Every search request needs to hit a copy of every shard in the index. That's fine if every shard is sitting on a different node, but not if many shards have to compete for the same resources.
- Term statistics, used to calculate relevance, are per shard. Having a small amount of data in many shards leads to poor relevance.

[TIP]

A little overallocation is good. A kagillion shards is bad. It is difficult to define what constitutes too many shards, as it depends on their size and how they are being used. A hundred shards that are seldom used may be fine, while two shards experiencing very heavy usage could be too many. Monitor your nodes to ensure that they have enough spare capacity to deal with exceptional conditions.

Scaling out should be done in phases. Build in enough capacity to get to the next phase. Once you get to the next phase, you have time to think about the changes you need to make to reach the phase after that.



[[capacity-planning]] === Capacity Planning

If 1 shard is too few and 1,000 shards are too many, how do I know how many shards I need?(((“shards”, “determining number you need”))))(((“capacity planning”))))(((“scaling”, “capacity planning”)))) This is a question that is impossible to answer in the general case. There are just too many variables: the hardware that you use, the size and complexity of your documents, how you index and analyze those documents, the types of queries that you run, the aggregations that you perform, how you model your data, and more.

Fortunately, it is an easy question to answer in the specific case--yours:

1. Create a cluster consisting of a single server, with the hardware that you are considering using in production.
2. Create an index with the same settings and analyzers that you plan to use in production, but with only one primary shard and no replicas.
3. Fill it with real documents (or as close to real as you can get).
4. Run real queries and aggregations (or as close to real as you can get).

Essentially, you want to replicate real-world usage and to push this single shard until it ``breaks.'' Even the definition of *breaks* depends on you: some users require that all responses return within 50ms; others are quite happy to wait for 5 seconds.

Once you define the capacity of a single shard, it is easy to extrapolate that number to your whole index. Take the total amount of data that you need to index, plus some extra for future growth, and divide by the capacity of a single shard. The result is the number of primary shards that you will need.

[TIP]

Capacity planning should not be your first step.

First look for ways to optimize how you are using Elasticsearch. Perhaps you have inefficient queries, not enough RAM, or you have left swap enabled?

We have seen new users who, frustrated by initial performance, immediately start trying to tune the garbage collector or adjust the number of threads, instead of tackling the simple problems like removing wildcard queries.

=====



[[replica-shards]] === Replica Shards

Up until now we have spoken only about primary shards, but we have another tool in our belt: replica shards.((("scaling", "replica shards")))((("shards", "replica")))((("replica shards"))))
The main purpose of replicas is for failover, as discussed in <>: if the node holding a primary shard dies, a replica is promoted to the role of primary.

At index time, a replica shard does the same amount of work as the primary shard. New documents are first indexed on the primary and then on any replicas. Increasing the number of replicas does not change the capacity of the index.

However, replica shards can serve read requests. If, as is often the case, your index is search heavy, you can increase search performance by increasing the number of replicas, but only if you also *add extra hardware*.

Let's return to our example of an index with two primary shards. We increased capacity of the index by adding a second node. Adding more nodes would not help us to add indexing capacity, but we could take advantage of the extra hardware at search time by increasing the number of replicas:

[source,json]

```
POST /my_index/_settings { "number_of_replicas": 1
```

```
}
```

Having two primary shards, plus a replica of each primary, would give us a total of four shards: one for each node, as shown in <>.

[[img-four-nodes]] .An index with two primary shards and one replica can scale out across four nodes image::images/elas_4403.png["An index with two primary shards and one replica can scale out across four nodes"]

==== Balancing Load with Replicas

Search performance depends on the response times of the slowest node, so it is a good idea to try to balance out the load across all nodes.((("replica shards", "balancing load with")))((("load balancing with replica shards")))) If we added just one extra node instead of two, we would end up with two nodes having one shard each, and one node doing double the work with two shards.

We can even things out by adjusting the number of replicas. By allocating two replicas instead of one, we end up with a total of six shards, which can be evenly divided between three nodes, as shown in <>:

[source,json]

```
POST /my_index/_settings { "number_of_replicas": 2
```

```
}
```

As a bonus, we have also increased our availability. We can now afford to lose two nodes and still have a copy of all our data.

[[img-three-nodes]] .Adjust the number of replicas to balance the load between nodes
image::images/elas_4404.png["Adjust the number of replicas to balance the load between nodes"]

NOTE: The fact that node 3 holds two replicas and no primaries is not important. Replicas and primaries do the same amount of work; they just play slightly different roles. There is no need to ensure that primaries are distributed evenly across all nodes.



[[multiple-indices]] === Multiple Indices

Finally, remember that there is no rule that limits your application to using only a single index.((("scaling", "using multiple indices")))(("indices", "multiple")) When we issue a search request, it is forwarded to a copy (a primary or a replica) of all the shards in an index. If we issue the same search request on multiple indices, the exact same thing happens--there are just more shards involved.

TIP: Searching 1 index of 50 shards is exactly equivalent to searching 50 indices with 1 shard each: both search requests hit 50 shards.

This can be a useful fact to remember when you need to add capacity on the fly. Instead of having to reindex your data into a bigger index, you can just do the following:

- Create a new index to hold new data.
- Search across both indices to retrieve new and old data.

In fact, with a little forethought, adding a new index can be done in a completely transparent way, without your application ever knowing that anything has changed.

In <>, we spoke about using an index alias to point to the current version of your index.((("index aliases")))(("aliases, index")) For instance, instead of naming your index `tweets`, name it `tweets_v1`. Your application would still talk to `tweets`, but in reality that would be an alias that points to `tweets_v1`. This allows you to switch the alias to point to a newer version of the index on the fly.

A similar technique can be used to expand capacity by adding a new index. It requires a bit of planning because you will need two aliases: one for searching and one for indexing:

[source,json]

```
PUT /tweets_1/_alias/tweets_search <1>
```

PUT /tweets_1/_alias/tweets_index <1>

<1> Both the `tweets_search` and the `tweets_index` alias point to index `tweets_1`.

New documents should be indexed into `tweets_index`, and searches should be performed against `tweets_search`. For the moment, these two aliases point to the same index.

When we need extra capacity, we can create a new index called `tweets_2` and update the aliases as follows:



[source,json]

```
POST /_aliases { "actions": [ { "add": { "index": "tweets_2", "alias": "tweets_search" }}, <1> {  
  "remove": { "index": "tweets_1", "alias": "tweets_index" }}, <2> { "add": { "index": "tweets_2",  
  "alias": "tweets_index" } } <2> ]  
}  
}
```

<1> Add index `tweets_2` to the `tweets_search` alias.

<2> Switch `tweets_index` from `tweets_1` to `tweets_2`.

A search request can target multiple indices, so having the search alias point to `tweets_1` and `tweets_2` is perfectly valid. However, indexing requests can target only a single index. For this reason, we have to switch the index alias to point to only the new index.

[TIP]

A document `GET` request, like(((("HTTP methods", "GET")))((("GET method")))) an indexing request, can target only one index. This makes retrieving a document by ID a bit more complicated in this scenario. Instead, run a search request with the <http://bit.ly/1C4Q0cf>[`ids` query], or do a(((("mget (multi-get) API"))) <http://bit.ly/1sDd2EX>[`multi-get`] request on `tweets_1` and `tweets_2`.

=====

Using multiple indices to expand index capacity on the fly is of particular benefit when dealing with time-based data such as logs or social-event streams, which we discuss in the next section.



[[time-based]] === Time-Based Data

One of the most common use cases for Elasticsearch is for logging,((("logging", "using Elasticsearch for")))(("time-based data")))((("scaling", "time-based data and"))) so common in fact that Elasticsearch provides an integrated((("ELK stack"))) logging platform called the *ELK stack*—Elasticsearch, Logstash, and Kibana--to make the process easy.

<http://www.elasticsearch.org/overview/logstash>[Logstash] collects, parses, and enriches logs before indexing them into Elasticsearch.((("Logstash"))) Elasticsearch acts as a centralized logging server, and <http://www.elasticsearch.org/overview/kibana>[Kibana] is a((("Kibana"))) graphic frontend that makes it easy to query and visualize what is happening across your network in near real-time.

Most traditional use cases for search engines involve a relatively static collection of documents that grows slowly. Searches look for the most relevant documents, regardless of when they were created.

Logging--and other time-based data streams such as social-network activity--are very different in nature. ((("social-network activity"))) The number of documents in the index grows rapidly, often accelerating with time. Documents are almost never updated, and searches mostly target the most recent documents. As documents age, they lose value.

We need to adapt our index design to function with the flow of time-based data.

[[index-per-timeframe]] ===== Index per Time Frame

If we were to have one big index for documents of this type, we would soon run out of space. Logging events just keep on coming, without pause or interruption. We could delete the old events, with a `delete-by-query` :

[source,json]

```
DELETE /logs/event/_query { "query": { "range": { "@timestamp": { <1> "lt": "now-90d" } } }}
```

```
}
```

<1> Deletes all documents where Logstash's `@timestamp` field is older than 90 days.

But this approach is *very inefficient*. Remember that when you delete a document, it is only *marked* as deleted (see <>). It won't be physically deleted until the segment containing it is merged away.



Instead, use an *index per time frame*. ((("indices", "index per-timeframe"))) You could start out with an index per year (`logs_2014`) or per month (`logs_2014-10`). Perhaps, when your website gets really busy, you need to switch to an index per day (`logs_2014-10-24`). Purging old data is easy: just delete old indices.

This approach has the advantage of allowing you to scale as and when you need to. You don't have to make any difficult decisions up front. Every day is a new opportunity to change your indexing time frames to suit the current demand. Apply the same logic to how big you make each index. Perhaps all you need is one primary shard per week initially. Later, maybe you need five primary shards per day. It doesn't matter--you can adjust to new circumstances at any time.

Aliases can help make switching indices more transparent.((("aliases", "index"))) For indexing, you can point `logs_current` to the index currently accepting new log events, and for searching, update `last_3_months` to point to all indices for the previous three months:

[source,json]

```
POST /_aliases { "actions": [ { "add": { "alias": "logs_current", "index": "logs_2014-10" }}, <1>
{ "remove": { "alias": "logs_current", "index": "logs_2014-09" }}, <1> { "add": { "alias":
"last_3_months", "index": "logs_2014-10" }}, <2> { "remove": { "alias": "last_3_months",
"index": "logs_2014-07" }}, <2> ] }
```

<1> Switch `logs_current` from September to October.

<2> Add October to `last_3_months` and remove July.



[[index-templates]] === Index Templates

Elasticsearch doesn't require you to create an index before using it.((("indices", "templates")))(("scaling", "index templates and")))((("templates", "index"))) With logging, it is often more convenient to rely on index autocreation than to have to create indices manually.

Logstash uses the timestamp((("Logstash")))(("timestamps, use by Logstash to create index names")) from an event to derive the index name. By default, it indexes into a different index every day, so an event with a `@timestamp` of `2014-10-01 00:00:01` will be sent to the index `logstash-2014.10.01`. If that index doesn't already exist, it will be created for us.

Usually we want some control over the settings and mappings of the new index. Perhaps we want to limit the number of shards to `1`, and we want to disable the `_all` field. Index templates can be used to control which settings should be applied to newly created indices:

[source,json]

```
PUT /template/my_logs <1> { "template": "logstash-*", <2> "order": 1, <3> "settings": {  
  "number_of_shards": 1 <4> }, "mappings": { "_default": { <5> "_all": { "enabled": false } } },  
  "aliases": { "last_3_months": {} <6> }
```

```
}
```

<1> Create a template called `my_logs`.

<2> Apply this template to all indices beginning with `logstash-`.

<3> This template should override the default `logstash` template that has a lower `order`.

<4> Limit the number of primary shards to `1`.

<5> Disable the `_all` field for all types.

<6> Add this index to the `last_3_months` alias.

This template specifies the default settings that will be applied to any index whose name begins with `logstash-`, whether it is created manually or automatically. If we think the index for tomorrow will need more capacity than today, we can update the index to use a higher number of shards.

The template even adds the newly created index into the `last_3_months` alias, although removing the old indices from that alias will have to be done manually.



[[retiring-data]] === Retiring Data

As time-based data ages, it becomes less relevant.((("scaling", "retiring data"))) It's possible that we will want to see what happened last week, last month, or even last year, but for the most part, we're interested in only the here and now.

The nice thing about an index per time frame ((("indices", "index per-timeframe", "deleting old data and")))((("indices", "deleting"))) is that it enables us to easily delete old data: just delete the indices that are no longer relevant:

[source,json]

DELETE /logs_2013*

Deleting a whole index is much more efficient than deleting individual documents: Elasticsearch just removes whole directories.

But deleting an index is very *final*. There are a number of things we can do to help data age gracefully, before we decide to delete it completely.

[[migrate-indices]] ===== Migrate Old Indices

With logging data, there is likely to be one *hot* index--the index for today.((("indices", "migrating old indices"))) All new documents will be added to that index, and almost all queries will target that index. It should use your best hardware.

How does Elasticsearch know which servers are your best servers? You tell it, by assigning arbitrary tags to each server. For instance, you could start a node as follows:

```
./bin/elasticsearch --node.box_type strong
```

The `box_type` parameter is completely arbitrary--you could have named it whatever you like--but you can use these arbitrary values to tell Elasticsearch where to allocate an index.

We can ensure that today's index is on our strongest boxes by creating it with the following settings:

[source,json]

```
PUT /logs_2014-10-01 { "settings": { "index.routing.allocation.include.box_type" : "strong" } }
```



Yesterday's index no longer needs to be on our strongest boxes, so we can move it to the nodes tagged as `medium` by updating its index settings:

[source,json]

```
POST /logs_2014-09-30/_settings { "index.routing.allocation.include.box_type" : "medium" }
```

```
}
```

[[optimize-indices]] ===== Optimize Indices

Yesterday's index is unlikely to change.((("indices", "optimizing"))) Log events are static: what happened in the past stays in the past. If we merge each shard down to just a single segment, it'll use fewer resources and will be quicker to query. We can do this with the <>.

It would be a bad idea to optimize the index while it was still allocated to the `strong` boxes, as the optimization process could swamp the I/O on those nodes and impact the indexing of today's logs. But the `medium` boxes aren't doing very much at all, so we are safe to optimize.

Yesterday's index may have replica shards.((("replica shards", "index optimization and"))) If we issue an optimize request, it will optimize the primary shard and the replica shards, which is a waste. Instead, we can remove the replicas temporarily, optimize, and then restore the replicas:

[source,json]

```
POST /logs_2014-09-30/_settings { "number_of_replicas": 0 }
```

```
POST /logs_2014-09-30/_optimize?max_num_segments=1
```

```
POST /logs_2014-09-30/_settings
```

```
{ "number_of_replicas": 1 }
```



Of course, without replicas, we run the risk of losing data if a disk suffers catastrophic failure.

You may((("snapshot-restore API"))) want to back up the data first, with the

<http://bit.ly/14ED13A>[`snapshot-restore` API].

[[close-indices]] ===== Closing Old Indices

As indices get even older, they reach a point where they are almost never accessed.

((("indices", "closing old indices"))) We could delete them at this stage, but perhaps you want to keep them around just in case somebody asks for them in six months.

These indices can be closed. They will still exist in the cluster, but they won't consume resources other than disk space. Reopening an index is much quicker than restoring it from backup.

Before closing, it is worth flushing the index to make sure that there are no transactions left in the transaction log. An empty transaction log will make index recovery faster when it is reopened:

[source,json]

```
POST /logs_2014-01-/flush <1> POST /logs_2014-01-/close <2>
```

POST /logs_2014-01-*/_open <3>

<1> Flush all indices from January to empty the transaction logs.

<2> Close all indices from January.

<3> When you need access to them again, reopen them with the `open` API.

[[archive-indices]] ===== Archiving Old Indices

Finally, very old indices ((("indices", "archiving old indices")))can be archived off to some long-term storage like a shared disk or Amazon's S3 using the

<http://bit.ly/14ED13A>[`snapshot-restore` API], just in case you may need to access them in the future. Once a backup exists, the index can be deleted from the cluster.



[[user-based]] === User-Based Data

Often, users start using Elasticsearch because they need to add full-text search or analytics to an existing application.((("scaling", "user-based data")))((("user-based data"))) They create a single index that holds all of their documents. Gradually, others in the company realize how much benefit Elasticsearch brings, and they want to add their data to Elasticsearch as well.

Fortunately, Elasticsearch supports [http://en.wikipedia.org/wiki/Multitenancy\[multitenancy\]](http://en.wikipedia.org/wiki/Multitenancy[multitenancy]) so each new user can have her own index in the same cluster.((("multitenancy"))) Occasionally, somebody will want to search across the documents for all users, which they can do by searching across all indices, but most of the time, users are interested in only their own documents.

Some users have more documents than others, and some users will have heavier search loads than others, so the ability to specify the number of primary shards and replica shards that each index should have fits well with the index-per-user model.((("indices", "index-per-user model")))((("primary shards", "number per-index"))) Similarly, busier indices can be allocated to stronger boxes with shard allocation filtering. (See <>.)

TIP: Don't just use the default number of primary shards for every index. Think about how much data that index needs to hold. It may be that all you need is one shard--any more is a waste of resources.

Most users of Elasticsearch can stop here. A simple index-per-user approach is sufficient for the majority of cases.

In exceptional cases, you may find that you need to support a large number of users, all with similar needs. An example might be hosting a search engine for thousands of email forums.((("forums", "resource allocation for"))) Some forums may have a huge amount of traffic, but the majority of forums are quite small. Dedicating an index with a single shard to a small forum is overkill--a single shard could hold the data for many forums.

What we need is a way to share resources across users, to give the impression that each user has his own index without wasting resources on small users.



[[shared-index]] === Shared Index

We can use a large shared index for the many smaller ((("scaling", "shared index"))) ((("indices", "shared"))) forums by indexing the forum identifier in a field and using it as a filter:

[source,json]

```
PUT /forums { "settings": { "number_of_shards": 10 <1> }, "mappings": { "post": { "properties": { "forum_id": { <2> "type": "string", "index": "not_analyzed" } } } } }
```

```
PUT /forums/post/1 { "forum_id": "baking", <2> "title": "Easy recipe for ginger nuts", ... }
```

```
}
```

<1> Create an index large enough to hold thousands of smaller forums.

<2> Each post must include a `forum_id` to identify which forum it belongs to.

We can use the `forum_id` as a filter to search within a single forum. The filter will exclude most of the documents in the index (those from other forums), and filter caching will ensure that responses are fast:

[source,json]

```
GET /forums/post/_search { "query": { "filtered": { "query": { "match": { "title": "ginger nuts" } }, "filter": { "term": { <1> "forum_id": { "baking" } } } } }
```

```
}
```

<1> The `term` filter is cached by default.

This approach works, but we can do better. ((("shards", "routing a document to"))) The posts from a single forum would fit easily onto one shard, but currently they are scattered across all ten shards in the index. This means that every search request has to be forwarded to a primary or replica of all ten shards. What would be ideal is to ensure that all the posts from a single forum are stored on the same shard.

In <>, we explained((("routing a document to a shard"))) that a document is allocated to a particular shard by using this formula:



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

The `routing` value defaults to the document's `_id`, but we can override that and provide our own custom routing value, such as `forum_id`. All documents with the same `routing` value will be stored on the same shard:

[source,json]

```
PUT /forums/post/1?routing=baking <1> { "forum_id": "baking", <1> "title": "Easy recipe for ginger nuts", ...
```

```
}
```

<1> Using `forum_id` as the routing value ensures that all posts from the same forum are stored on the same shard.

When we search for posts in a particular forum, we can pass the same `routing` value to ensure that the search request is run on only the single shard that holds our documents:

[source,json]

```
GET /forums/post/_search?routing=baking <1> { "query": { "filtered": { "query": { "match": { "title": "ginger nuts" } }, "filter": { "term": { <2> "forum_id": { "baking" } } } } }
```

```
}
```

<1> The query is run on only the shard that corresponds to this `routing` value.

<2> We still need the filter, as a single shard can hold posts from many forums.

Multiple forums can be queried by passing a comma-separated list of `routing` values, and including each `forum_id` in a `terms` filter:

[source,json]



```
GET /forums/post/_search?routing=baking,cooking,recipes { "query": { "filtered": { "query": { "match": { "title": "ginger nuts" } }, "filter": { "terms": { "forum_id": [ "baking", "cooking", "recipes" ] } } } } }
```

```
}
```

While this approach is technically efficient, it looks a bit clumsy because of the need to specify `routing` values and `terms` filters on every query or indexing request. Index aliases to the rescue!



[[faking-it]] === Faking Index per User with Aliases

To keep our design simple and clean, we would ((("scaling", "faking index-per-user with aliases"))) ((("aliases, index"))) ((("index aliases"))) like our application to believe that we have a dedicated index per user--or per forum in our example--even if the reality is that we are using one big <1>. To do that, we need some way to hide the `routing` value and the filter on `forum_id`.

Index aliases allow us to do just that. When you associate an alias with an index, you can also specify a filter and routing values:

[source,json]

```
PUT /forums/_alias/baking { "routing": "baking", "filter": { "term": { "forum_id": "baking" } } }
```

```
}
```

Now, we can treat the `baking` alias as if it were its own index. Documents indexed into the `baking` alias automatically get the custom routing value applied:

[source,json]

```
PUT /baking/post/1 <1> { "forum_id": "baking", <1> "title": "Easy recipe for ginger nuts", ... }
```

```
}
```

<1> We still need the `forum_id` field for the filter to work, but the custom routing value is now implicit.

Queries run against the `baking` alias are run just on the shard associated with the custom routing value, and the results are automatically filtered by the filter we specified:

[source,json]

```
GET /baking/post/_search { "query": { "match": { "title": "ginger nuts" } } }
```

```
}
```



Multiple aliases can be specified when searching across multiple forums:

[source,json]

```
GET /baking,recipes/post/_search <1> { "query": { "match": { "title": "ginger nuts" } } }
```

```
}
```

<1> Both `routing` values are applied, and results can match either filter.



[[one-big-user]] === One Big User

Big, popular forums start out as small forums.((("forums", "resource allocation for", "one big user"))) One day we will find that one shard in our shared index is doing a lot more work than the other shards, because it holds the documents for a forum that has become very popular. That forum now needs its own index.

The index aliases that we're using to fake an index per user give us a clean migration path for the big forum.((("indices", "shared", "migrating data to dedicated index")))

The first step is to create a new index dedicated to the forum, and with the appropriate number of shards to allow for expected growth:

[source,json]

```
PUT /baking_v1 { "settings": { "number_of_shards": 3 } }
```

```
}
```

The next step is to migrate the data from the shared index into the dedicated index, which can be done using <> and the <>. As soon as the migration is finished, the index alias can be updated to point to the new index:

[source,json]

```
POST/_aliases { "actions": [ { "remove": { "alias": "baking", "index": "forums" }}, { "add": { "alias": "baking", "index": "baking_v1" }} ] }
```

```
}
```

Updating the alias is atomic; it's like throwing a switch. Your application continues talking to the `baking` API and is completely unaware that it now points to a new dedicated index.

The dedicated index no longer needs the filter or the routing values. We can just rely on the default sharding that Elasticsearch does using each document's `_id` field.

The last step is to remove the old documents from the shared index, which can be done with a `delete-by-query` request, using the original routing value and forum ID:



[source,json]

```
DELETE /forums/post/_query?routing=baking { "query": { "term": { "forum_id": "baking" } } }
```

```
}
```

The beauty of this index-per-user model is that it allows you to reduce resources, keeping costs low, while still giving you the flexibility to scale out when necessary, and with zero downtime.



[[finite-scale]] === Scale Is Not Infinite

Throughout this chapter we have spoken about many of the ways that Elasticsearch can scale. ((("scaling", "scale is not infinite"))) Most scaling problems can be solved by adding more nodes. But one resource is finite and should be treated with respect: the cluster state. ((("cluster state")))

The *cluster state* is a data structure that holds the following cluster-level information:

- Cluster-level settings
- Nodes that are part of the cluster
- Indices, plus their settings, mappings, analyzers, warmers, and aliases
- The shards associated with each index, plus the node on which they are allocated

You can view the current cluster state with this request:

[source,json]

GET /_cluster/state

The cluster state exists on every node in the cluster, ((("nodes", "cluster state"))) including client nodes. This is how any node can forward a request directly to the node that holds the requested data--every node knows where every document lives.

Only the master node is allowed to update the cluster state. Imagine that an indexing request introduces a previously unknown field. The node holding the primary shard for the document must forward the new mapping to the master node. The master node incorporates the changes in the cluster state, and publishes a new version to all of the nodes in the cluster.

Search requests *use* the cluster state, but they don't change it. The same applies to document-level CRUD requests unless, of course, they introduce a new field that requires a mapping update. By and large, the cluster state is static and is not a bottleneck.

However, remember that this same data structure has to exist in memory on every node, and must be published to every node whenever it is updated. The bigger it is, the longer that process will take.

The most common problem that we see with the cluster state is the introduction of too many fields. A user might decide to use a separate field for every IP address, or every referer URL. The following example keeps track of the number of times a page has been visited by using a different field name for every unique referer:



[role="pagebreak-before"]

[source,json]

```
POST /counters/pageview/home_page/_update { "script": "ctx._source[referer]++", "params":  
{ "referer": "http://www.foo.com/links?bar=baz" } }
```

}

This approach is catastrophically bad! It will result in millions of fields, all of which have to be stored in the cluster state. Every time a new referer is seen, a new field is added to the already bloated cluster state, which then has to be published to every node in the cluster.

A much better approach (((“nested objects”))))((("objects", "nested"))))is to use <>, with one field for the parameter name— `referer` and another field for its associated value
— `count` :

[source,json]

```
"counters": [  
  { "referer": "http://www.foo.com/links?bar=baz", "count": 2 },  
  { "referer": "http://www.linkbait.com/article_3", "count": 10 },  
  ...  
]
```

The nested approach may increase the number of documents, but Elasticsearch is built to handle that. The important thing is that it keeps the cluster state small and agile.

Eventually, despite your best intentions, you may find that the number of nodes and indices and mappings that you have is just too much for one cluster. At this stage, it is probably worth dividing the problem into multiple clusters. Thanks to

<http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/modules-tribe.html>[`tribe` nodes], you can even run searches across multiple clusters, as if they were one big cluster.



[[hardware]] === Hardware

如果你已经遵循了正常的开发路径, 你可能已经在自己的笔记本电脑或周边的机器集群上使用了 Elasticsearch (((("deployment", "hardware")))((("hardware")))). 但当要部署到生产环境时, 有一些建议是你需要考虑的.

这里没有什么必须要遵守的准则; Elasticsearch 被用于在众多的机器上处理各种任务. 但基于我们的生产集群经验, 可以提供一个好的起点.

==== 内存

如果有一种资源是最先被耗尽的, 它可能是内存. (((("hardware", "memory")))((("memory")))) 排序和聚合都是很耗内存的, 所以使用足够的堆空间来应付它们是很重要的. (((("heap")))) 尽管当堆空间是比较小的时候, 也能为操作系统文件缓存提供额外的内存. 因为 Lucene 使用的很多数据结构格式是基于硬盘的, Elasticsearch 使得操作系统缓存能产生很大效果.

64 GB 内存的机器是非常理想的, 但 32 GB 和 16 GB 机器也是很常见的. 少于 8 GB 会适得其反 (你最终需要很多很多的小机器), 大于 64 GB 的机器也会有问题, 我们将在 <> 中讨论.

==== CPUs

大多数 Elasticsearch 部署对 CPU 要求很小. 例如, (((("CPUs (central processing units)")))) (((("hardware", "CPUs")))) 额外的处理器安装问题比其它资源要少. 你应该选择多核的现代处理器. 常规的集群利用 2 到 8 核机器.

如果你要在更快的 CUPs 和更多的核心之间选择, 选择更多的核心更好. 多核心提供的额外并发将远远超出稍微快点的时钟速度(注: CPU 速度).

==== 硬盘

硬盘对所有的集群都很重要, (((("disks")))) (((("hardware", "disks")))) 对高度索引的集群更是加倍重要 (例如那些存储日志数据的). 硬盘是服务器上最慢的子系统, 这意味着那些写多的集群很容易让硬盘饱和, 使得它成为集群的瓶颈.

如果你负担得起 SSD, 它将远远超出任何旋转媒介(注: 机械硬盘, 磁带等). 基于 SSD 的节点的查询和索引性能都有提升. 如果你负担得起, SSD 是一个好的选择.

. 检查你的 I/O 调度程序

如果你正在使用 SSDs, 确保你的系统 I/O 调度程序是 (((("I/O scheduler")))) 配置正确的. 当你向硬盘写数据, I/O 调度程序决定何时把数据实际发送到硬盘. 大多数 *nix 发行版下的调度程序都叫做 `cfs` (Completely Fair Queuing).

调度程序分配时间片到每个进程, 并且优化这些到硬盘的众多队列的传递. 但它是为旋转媒介优化的: 旋转盘片的固有特性意味着它写入数据到基于物理布局的硬盘会更高效.



这对SSD来说是低效的,然而,尽管这里没有涉及到旋转盘片.但是, `deadline` 或 `noop` 应该被使用.`deadline` 调度程序基于写入等待时间进行优化, `noop` 只是一个简单的FIFO队列.

这个简单的更改可以带来显著的影响.仅仅是使用正确的调度程序,我们看到了500倍的写入能力提升.

如果你使用旋转媒介,尝试获取尽可能快的硬盘(高性能服务器硬盘, 15k RPM 驱动器).

使用RAID 0是提高硬盘速度的有效途径,对旋转硬盘和SSD来说都是如此.

没有必要使用镜像或其它RAID变体,因为高可用已经通过replicas内建于Elasticsearch之中.

最后,避免使用网络附加存储(NAS).人们常声称他们的NAS解决方案比本地驱动器更快更可靠.除却这些声称,我们从没看到NAS能配得上它的大肆宣传.NAS常常很慢,显露出更大的延时和更宽的平均延时方差,而且它是单点故障的.

===== 网络

快速可靠的网络显然对分布式系统的性能是很重要的(((`"hardware", "network"`)))
((`"network"`))).

低延时能帮助确保节点间能容易的通讯,大带宽能帮助分片移动和恢复.现代数据中心网络(1 GbE, 10 GbE)对绝大多数集群都是足够的.

即使数据中心们近在咫尺,也要避免集群跨越多个数据中心.绝对要避免集群跨越大的地理距离.

Elasticsearch假定所有节点都是平等的--并不会因为有一半的节点在150ms外的另一数据中心而有所不同.更大的延时会加重分布式系统中的问题而且使得调试和排错更困难.

和NAS的争论类似,每个人都声称他们的数据中心间的线路都是健壮和低延时的.这是真的--直到它不是时(网络失败终究是会发生的;你可以相信它).从我们的经验来看,处理跨数据中心集群的麻烦事--是根本不值得的.

===== 一般注意事项

获取真正的巨型机器在今天是可能的:(((`"hardware", "general considerations"`))) 成百GB的RAM 和几十个CPU核心.

反之,在云平台上串联起成千的小虚拟机也是可能的,例如 EC2(注:Amazon Elastic Compute Cloud).

哪条道路是最好的?

通常,选择中到大型机器更好.避免使用小型机器,因为你不会希望去管理拥有上千个节点的集群,而且在这些小型机器上运行Elasticsearch的开销也是显著的.

与此同时,避免使用真正的巨型机器.它们通常会导致资源使用不均衡(例如,所有的内存都被使用,但CPU没有)而且在单机上运行多个节点时,会增加逻辑复杂度.





==== Java Virtual Machine

You should always run the most recent version of the Java Virtual Machine (JVM), unless otherwise stated on the Elasticsearch website.((("deployment", "Java Virtual Machine (JVM")")))((("JVM (Java Virtual Machine")")))((("Java Virtual Machine", see="JVM"))))
Elasticsearch, and in particular Lucene, is a demanding piece of software. The unit and integration tests from Lucene often expose bugs in the JVM itself. These bugs range from mild annoyances to serious segfaults, so it is best to use the latest version of the JVM where possible.

Java 7 is strongly preferred over Java 6. Either Oracle or OpenJDK are acceptable. They are comparable in performance and stability.

If your application is written in Java and you are using the transport client or node client, make sure the JVM running your application is identical to the server JVM. In few locations in Elasticsearch, Java's native serialization is used (IP addresses, exceptions, and so forth). Unfortunately, Oracle has been known to change the serialization format between minor releases, leading to strange errors. This happens rarely, but it is best practice to keep the JVM versions identical between client and server.

.Please Do Not Tweak JVM Settings

The JVM exposes dozens (hundreds even!) of settings, parameters, and configurations.((("JVM (Java Virtual Machine)", "avoiding custom configuration")))) They allow you to tweak and tune almost every aspect of the JVM.

When a knob is encountered, it is human nature to want to turn it. We implore you to squash this desire and *not* use custom JVM settings. Elasticsearch is a complex piece of software, and the current JVM settings have been tuned over years of real-world usage.

It is easy to start turning knobs, producing opaque effects that are hard to measure, and eventually detune your cluster into a slow, unstable mess. When debugging clusters, the first step is often to remove all custom configurations. About half the time, this alone restores stability and performance.

==== Transport Client Versus Node Client

If you are using Java, you may wonder when to use the transport client versus the node client.((("Java", "clients for Elasticsearch"))))((("clients"))))((("node client", "versus transport client"))))((("transport client", "versus node client")))) As discussed at the beginning of the



book, the transport client acts as a communication layer between the cluster and your application. It knows the API and can automatically round-robin between nodes, sniff the cluster for you, and more. But it is *external* to the cluster, similar to the REST clients.

The node client, on the other hand, is actually a node within the cluster (but does not hold data, and cannot become master). Because it is a node, it knows the entire cluster state (where all the nodes reside, which shards live in which nodes, and so forth). This means it can execute APIs with one less network hop.

There are uses-cases for both clients:

- The transport client is ideal if you want to decouple your application from the cluster. For example, if your application quickly creates and destroys connections to the cluster, a transport client is much "lighter" than a node client, since it is not part of a cluster. + Similarly, if you need to create thousands of connections, you don't want to have thousands of node clients join the cluster. The TC will be a better choice.
- On the flipside, if you need only a few long-lived, persistent connection objects to the cluster, a node client can be a bit more efficient since it knows the cluster layout. But it ties your application into the cluster, so it may pose problems from a firewall perspective.

==== Configuration Management

If you use configuration management already (Puppet, Chef, Ansible), you can skip this tip.
((("deployment", "configuration management")))((("configuration management"))))

If you don't use configuration management tools yet, you should! Managing a handful of servers by `parallel-ssh` may work now, but it will become a nightmare as you grow your cluster. It is almost impossible to edit 30 configuration files by hand without making a mistake.

Configuration management tools help make your cluster consistent by automating the process of config changes. It may take a little time to set up and learn, but it will pay itself off handsomely over time.



==== Important Configuration Changes Elasticsearch ships with *very good* defaults, (((("deployment", "configuration changes, important"))))(((("configuration changes, important")))) especially when it comes to performance- related settings and options. When in doubt, just leave the settings alone. We have witnessed countless dozens of clusters ruined by errant settings because the administrator thought he could turn a knob and gain 100-fold improvement.

[NOTE]

Please read this entire section! All configurations presented are equally important, and are not listed in any particular order. Please read

through all configuration options and apply them to your cluster.

Other databases may require tuning, but by and large, Elasticsearch does not. If you are hitting performance problems, the solution is usually better data layout or more nodes. There are very few "magic knobs" in Elasticsearch. If there were, we'd have turned them already!

With that said, there are some *logistical* configurations that should be changed for production. These changes are necessary either to make your life easier, or because there is no way to set a good default (because it depends on your cluster layout).

===== Assign Names

Elasticsearch by default starts a cluster named `elasticsearch` . (((("configuration changes, important", "assigning names")))) It is wise to rename your production cluster to something else, simply to prevent accidents whereby someone's laptop joins the cluster. A simple change to `elasticsearch_production` can save a lot of heartache.

This can be changed in your `elasticsearch.yml` file:

[source,yaml]

cluster.name: elasticsearch_production



Similarly, it is wise to change the names of your nodes. As you've probably noticed by now, Elasticsearch assigns a random Marvel superhero name to your nodes at startup. This is cute in development--but less cute when it is 3a.m. and you are trying to remember which physical machine was Tagak the Leopard Lord.

More important, since these names are generated on startup, each time you restart your node, it will get a new name. This can make logs confusing, since the names of all the nodes are constantly changing.

Boring as it might be, we recommend you give each node a name that makes sense to you--a plain, descriptive name. This is also configured in your `elasticsearch.yml` :

[source,yaml]

node.name: elasticsearch_005_data

==== Paths

By default, Elasticsearch will place the plug-ins,(("configuration changes, important", "paths"))(("paths"))) logs, and--most important--your data in the installation directory. This can lead to unfortunate accidents, whereby the installation directory is accidentally overwritten by a new installation of Elasticsearch. If you aren't careful, you can erase all your data.

Don't laugh--we've seen it happen more than a few times.

The best thing to do is relocate your data directory outside the installation location. You can optionally move your plug-in and log directories as well.

This can be changed as follows:

[source,yaml]

`path.data: /path/to/data1,/path/to/data2 <1>`

Path to log files:

`path.logs: /path/to/logs`



Path to where plugins are installed:

path.plugins: /path/to/plugins

<1> Notice that you can specify more than one directory for data by using comma-separated lists.

Data can be saved to multiple directories, and if each directory is mounted on a different hard drive, this is a simple and effective way to set up a software RAID 0. Elasticsearch will automatically stripe data between the different directories, boosting performance

==== Minimum Master Nodes

The `minimum_master_nodes` setting is *extremely* important to the stability of your cluster.

((("configuration changes, important", "minimummaster_nodes setting")))

((("minimum_master_nodes setting"))) This setting helps prevent _split brains, the existence of two masters in a single cluster.

When you have a split brain, your cluster is at danger of losing data. Because the master is considered the supreme ruler of the cluster, it decides when new indices can be created, how shards are moved, and so forth. If you have *two* masters, data integrity becomes perilous, since you have two nodes that think they are in charge.

This setting tells Elasticsearch to not elect a master unless there are enough master-eligible nodes available. Only then will an election take place.

This setting should always be configured to a quorum (majority) of your master-eligible nodes.((("quorum"))) A quorum is `(number of master-eligible nodes / 2) + 1`. Here are some examples:

- If you have ten regular nodes (can hold data, can become master), a quorum is `6`.
- If you have three dedicated master nodes and a hundred data nodes, the quorum is `2`, since you need to count only nodes that are master eligible.
- If you have two regular nodes, you are in a conundrum. A quorum would be `2`, but this means a loss of one node will make your cluster inoperable. A setting of `1` will allow your cluster to function, but doesn't protect against split brain. It is best to have a minimum of three nodes in situations like this.

This setting can be configured in your `elasticsearch.yml` file:

[source,yaml]



discovery.zen.minimum_master_nodes: 2

But because Elasticsearch clusters are dynamic, you could easily add or remove nodes that will change the quorum. It would be extremely irritating if you had to push new configurations to each node and restart your whole cluster just to change the setting.

For this reason, `minimum_master_nodes` (and other settings) can be configured via a dynamic API call. You can change the setting while your cluster is online:

[source,js]

```
PUT /_cluster/settings { "persistent" : { "discovery.zen.minimum_master_nodes" : 2 } }
```

```
}
```

This will become a persistent setting that takes precedence over whatever is in the static configuration. You should modify this setting whenever you add or remove master-eligible nodes.

==== Recovery Settings

Several settings affect the behavior of shard recovery when your cluster restarts.(((("recovery settings"))))((("configuration changes, important", "recovery settings"))) First, we need to understand what happens if nothing is configured.

Imagine you have ten nodes, and each node holds a single shard--either a primary or a replica--in a 5 primary / 1 replica index. You take your entire cluster offline for maintenance (installing new drives, for example). When you restart your cluster, it just so happens that five nodes come online before the other five.

Maybe the switch to the other five is being flaky, and they didn't receive the restart command right away. Whatever the reason, you have five nodes online. These five nodes will gossip with each other, elect a master, and form a cluster. They notice that data is no longer evenly distributed, since five nodes are missing from the cluster, and immediately start replicating new shards between each other.

Finally, your other five nodes turn on and join the cluster. These nodes see that *their* data is being replicated to other nodes, so they delete their local data (since it is now redundant, and may be outdated). Then the cluster starts to rebalance even more, since the cluster size just went from five to ten.



During this whole process, your nodes are thrashing the disk and network, moving data around--for no good reason. For large clusters with terabytes of data, this useless shuffling of data can take a *really long time*. If all the nodes had simply waited for the cluster to come online, all the data would have been local and nothing would need to move.

Now that we know the problem, we can configure a few settings to alleviate it. First, we need to give Elasticsearch a hard limit:

[source,yaml]

gateway.recover_after_nodes: 8

This will prevent Elasticsearch from starting a recovery until at least eight (data or master) nodes are present. The value for this setting is a matter of personal preference: how many nodes do you want present before you consider your cluster functional? In this case, we are setting it to `8`, which means the cluster is inoperable unless there are at least eight nodes.

Then we tell Elasticsearch how many nodes *should* be in the cluster, and how long we want to wait for all those nodes:

[source,yaml]

```
gateway.expected_nodes: 10
```

gateway.recover_after_time: 5m

What this means is that Elasticsearch will do the following:

- Wait for eight nodes to be present
- Begin recovering after 5 minutes *or* after ten nodes have joined the cluster, whichever comes first.

These three settings allow you to avoid the excessive shard swapping that can occur on cluster restarts. It can literally make recovery take seconds instead of hours.

NOTE: These settings can only be set in the `config/elasticsearch.yml` file or on the command line (they are not dynamically updatable) and they are only relevant during a full cluster restart.

==== Prefer Unicast over Multicast



Elasticsearch is configured to use multicast discovery out of the box.

Multicast(((“configuration changes, important”, “preferring unicast over multicast”)))
(((“unicast, preferring over multicast”)))(((“multicast versus unicast”))) works by sending UDP pings across your local network to discover nodes. Other Elasticsearch nodes will receive these pings and respond. A cluster is formed shortly after.

Multicast is excellent for development, since you don't need to do anything. Turn a few nodes on, and they automatically find each other and form a cluster.

This ease of use is the exact reason you should disable it in production. The last thing you want is for nodes to accidentally join your production network, simply because they received an errant multicast ping. There is nothing wrong with multicast *per se*. Multicast simply leads to silly problems, and can be a bit more fragile (for example, a network engineer fiddles with the network without telling you--and all of a sudden nodes can't find each other anymore).

In production, it is recommended to use unicast instead of multicast. This works by providing Elasticsearch a list of nodes that it should try to contact. Once the node contacts a member of the unicast list, it will receive a full cluster state that lists all nodes in the cluster. It will then proceed to contact the master and join.

This means your unicast list does not need to hold all the nodes in your cluster. It just needs enough nodes that a new node can find someone to talk to. If you use dedicated masters, just list your three dedicated masters and call it a day. This setting is configured in your

```
elasticsearch.yml :
```

[source,yaml]

```
discovery.zen.ping.multicast.enabled: false <1>
```

discovery.zen.ping.unicast.hosts: ["host1", "host2:port"]

<1> Make sure you disable multicast, since it can operate in parallel with unicast.



==== Don't Touch These Settings!

There are a few hotspots in Elasticsearch that people just can't seem to avoid tweaking.
((("deployment", "settings to leave unaltered")))) We understand: knobs just beg to be turned. But of all the knobs to turn, these you should *really* leave alone. They are often abused and will contribute to terrible stability or terrible performance. Or both.

===== Garbage Collector

As briefly introduced in <>, the JVM uses a garbage collector to free unused memory.
((("garbage collector")))) This tip is really an extension of the last tip, but deserves its own section for emphasis:

Do not change the default garbage collector!

The default GC for Elasticsearch is Concurrent-Mark and Sweep (CMS).((("Concurrent-Mark and Sweep (CMS) garbage collector")))) This GC runs concurrently with the execution of the application so that it can minimize pauses. It does, however, have two stop-the-world phases. It also has trouble collecting large heaps.

Despite these downsides, it is currently the best GC for low-latency server software like Elasticsearch. The official recommendation is to use CMS.

There is a newer GC called the Garbage First GC (G1GC). ((("Garbage First GC (G1GC)")))) This newer GC is designed to minimize pausing even more than CMS, and operate on large heaps. It works by dividing the heap into regions and predicting which regions contain the most reclaimable space. By collecting those regions first (*garbage first*), it can minimize pauses and operate on very large heaps.

Sounds great! Unfortunately, G1GC is still new, and fresh bugs are found routinely. These bugs are usually of the segfault variety, and will cause hard crashes. The Lucene test suite is brutal on GC algorithms, and it seems that G1GC hasn't had the kinks worked out yet.

We would like to recommend G1GC someday, but for now, it is simply not stable enough to meet the demands of Elasticsearch and Lucene.

===== Threadpools

Everyone *loves* to tweak threadpools.((("threadpools")))) For whatever reason, it seems people cannot resist increasing thread counts. Indexing a lot? More threads! Searching a lot? More threads! Node idling 95% of the time? More threads!

The default threadpool settings in Elasticsearch are very sensible. For all threadpools (except `search`) the threadcount is set to the number of CPU cores. If you have eight cores, you can be running only eight threads simultaneously. It makes sense to assign only eight threads to any particular threadpool.



Search gets a larger threadpool, and is configured to `# cores * 3`.

You might argue that some threads can block (such as on a disk I/O operation), which is why you need more threads. This is not a problem in Elasticsearch: much of the disk I/O is handled by threads managed by Lucene, not Elasticsearch.

Furthermore, threadpools cooperate by passing work between each other. You don't need to worry about a networking thread blocking because it is waiting on a disk write. The networking thread will have long since handed off that work unit to another threadpool and gotten back to networking.

Finally, the compute capacity of your process is finite. Having more threads just forces the processor to switch thread contexts. A processor can run only one thread at a time, so when it needs to switch to a different thread, it stores the current state (registers, and so forth) and loads another thread. If you are lucky, the switch will happen on the same core. If you are unlucky, the switch may migrate to a different core and require transport on an inter-core communication bus.

This context switching eats up cycles simply by doing administrative housekeeping; estimates can peg it as high as 30 μ s on modern CPUs. So unless the thread will be blocked for longer than 30 μ s, it is highly likely that that time would have been better spent just processing and finishing early.

People routinely set threadpools to silly values. On eight core machines, we have run across configs with 60, 100, or even 1000 threads. These settings will simply thrash the CPU more than getting real work done.

So. Next time you want to tweak a threadpool, please don't. And if you *absolutely cannot resist*, please keep your core count in mind and perhaps set the count to double. More than that is just a waste.



[[heap-sizing]] === Heap: Sizing and Swapping

The default installation of Elasticsearch is configured with a 1 GB heap. (((("deployment", "heap", "sizing and swapping")))((("heap", "sizing and setting"))))) For just about every deployment, this number is far too small. If you are using the default heap values, your cluster is probably configured incorrectly.

There are two ways to change the heap size in Elasticsearch. The easiest is to set an environment variable called `ES_HEAP_SIZE` .(((("ES_HEAP_SIZE environment variable")))) When the server process starts, it will read this environment variable and set the heap accordingly. As an example, you can set it via the command line as follows:

[source,bash]

```
export ES_HEAP_SIZE=10g
```

Alternatively, you can pass in the heap size via a command-line argument when starting the process, if that is easier for your setup:

[source,bash]

```
./bin/elasticsearch -Xmx10g -Xms10g <1>
```

<1> Ensure that the min (`xms`) and max (`xmx`) sizes are the same to prevent the heap from resizing at runtime, a very costly process.

Generally, setting the `ES_HEAP_SIZE` environment variable is preferred over setting explicit `-Xmx` and `-Xms` values.

===== Give Half Your Memory to Lucene

A common problem is configuring a heap that is *too* large. (((("heap", "sizing and setting", "giving half your memory to Lucene")))) You have a 64 GB machine--and by golly, you want to give Elasticsearch all 64 GB of memory. More is better!

Heap is definitely important to Elasticsearch. It is used by many in-memory data structures to provide fast operation. But with that said, there is another major user of memory that is *off heap*: Lucene.



Lucene is designed to leverage the underlying OS for caching in-memory data structures. (((("Lucene", "memory for"))) Lucene segments are stored in individual files. Because segments are immutable, these files never change. This makes them very cache friendly, and the underlying OS will happily keep hot segments resident in memory for faster access.

Lucene's performance relies on this interaction with the OS. But if you give all available memory to Elasticsearch's heap, there won't be any left over for Lucene. This can seriously impact the performance of full-text search.

The standard recommendation is to give 50% of the available memory to Elasticsearch heap, while leaving the other 50% free. It won't go unused; Lucene will happily gobble up whatever is left over.

[[compressed_oops]] ===== Don't Cross 32 GB! There is another reason to not allocate enormous heaps to Elasticsearch. As it turns(((("heap", "sizing and setting", "32gb heap boundary")))((("32gb Heap boundary")))) out, the JVM uses a trick to compress object pointers when heaps are less than ~32 GB.

In Java, all objects are allocated on the heap and referenced by a pointer. Ordinary object pointers (OOP) point at these objects, and are traditionally the size of the CPU's native *word*: either 32 bits or 64 bits, depending on the processor. The pointer references the exact byte location of the value.

For 32-bit systems, this means the maximum heap size is 4 GB. For 64-bit systems, the heap size can get much larger, but the overhead of 64-bit pointers means there is more wasted space simply because the pointer is larger. And worse than wasted space, the larger pointers eat up more bandwidth when moving values between main memory and various caches (LLC, L1, and so forth).

Java uses a trick called

[object offsets.\(\(\(\("object offsets"\)\)\)\) This means a 32-bit pointer can reference four billion *objects*, rather than four billion bytes. Ultimately, this means the heap can grow to around 32 GB of physical size while still using a 32-bit pointer.](https://wikis.oracle.com/display/HotSpotInternals/CompressedOops[compressed oops]((()

Once you cross that magical ~30–32 GB boundary, the pointers switch back to ordinary object pointers. The size of each pointer grows, more CPU-memory bandwidth is used, and you effectively lose memory. In fact, it takes until around 40–50 GB of allocated heap before you have the same *effective* memory of a 32 GB heap using compressed oops.



The moral of the story is this: even when you have memory to spare, try to avoid crossing the 32 GB heap boundary. It wastes memory, reduces CPU performance, and makes the GC struggle with large heaps.

[role="pagebreak-before"] .I Have a Machine with 1 TB RAM!

The 32 GB line is fairly important. So what do you do when your machine has a lot of memory? It is becoming increasingly common to see super-servers with 300–500 GB of RAM.

First, we would recommend avoiding such large machines (see <>).

But if you already have the machines, you have two practical options:

- Are you doing mostly full-text search? Consider giving 32 GB to Elasticsearch and letting Lucene use the rest of memory via the OS filesystem cache. All that memory will cache segments and lead to blisteringly fast full-text search.
 - Are you doing a lot of sorting/aggregations? You'll likely want that memory in the heap then. Instead of one node with 32 GB+ of RAM, consider running two or more nodes on a single machine. Still adhere to the 50% rule, though. So if your machine has 128 GB of RAM, run two nodes, each with 32 GB. This means 64 GB will be used for heaps, and 64 will be left over for Lucene. + If you choose this option, set `cluster.routing.allocation.same_shard.host: true` in your config. This will prevent a primary and a replica shard from colocating to the same physical machine (since this would remove the benefits of replica high availability).
-

===== Swapping Is the Death of Performance

It should be obvious,(((("heap", "sizing and setting", "swapping, death of performance")))((("memory", "swapping as the death of performance")))((("swapping, the death of performance")))) but it bears spelling out clearly: swapping main memory to disk will *crush* server performance. Think about it: an in-memory operation is one that needs to execute quickly.

If memory swaps to disk, a 100-microsecond operation becomes one that take 10 milliseconds. Now repeat that increase in latency for all other 10us operations. It isn't difficult to see why swapping is terrible for performance.

The best thing to do is disable swap completely on your system. This can be done temporarily:



[source,bash]

sudo swapoff -a

To disable it permanently, you'll likely need to edit your `/etc/fstab`. Consult the documentation for your OS.

If disabling swap completely is not an option, you can try to lower `swappiness`. This value controls how aggressively the OS tries to swap memory. This prevents swapping under normal circumstances, but still allows the OS to swap under emergency memory situations.

For most Linux systems, this is configured using the `sysctl` value:

[source,bash]

vm.swappiness = 1 <1>

<1> A `swappiness` of `1` is better than `0`, since on some kernel versions a `swappiness` of `0` can invoke the OOM-killer.

Finally, if neither approach is possible, you should enable `mlockall`. file. This allows the JVM to lock its memory and prevent it from being swapped by the OS. In your `elasticsearch.yml`, set this:

[source,yaml]

bootstrap.mlockall: true



==== File Descriptors and MMap

Lucene uses a *very* large number of files. (((("deployment", "file descriptors and MMap")))) At the same time, Elasticsearch uses a large number of sockets to communicate between nodes and HTTP clients. All of this requires available file descriptors.((("file descriptors"))))

Sadly, many modern Linux distributions ship with a paltry 1,024 file descriptors allowed per process. This is *far* too low for even a small Elasticsearch node, let alone one that is handling hundreds of indices.

You should increase your file descriptor count to something very large, such as 64,000. This process is irritatingly difficult and highly dependent on your particular OS and distribution. Consult the documentation for your OS to determine how best to change the allowed file descriptor count.

Once you think you've changed it, check Elasticsearch to make sure it really does have enough file descriptors:

[source,js]

```
GET /_nodes/process
```

```
{ "cluster_name": "elasticsearch_zach", "nodes": { "TGn9iO2_QQKb0kavcLbnDw": {  
    "name": "Zach", "transport_address": "inet[/192.168.1.131:9300]", "host": "zacharys-air", "ip":  
    "192.168.1.131", "version": "2.0.0-SNAPSHOT", "build": "612f461", "http_address":  
    "inet[/192.168.1.131:9200]", "process": { "refresh_interval_in_millis": 1000, "id": 19808,  
    "max_file_descriptors": 64000, <1> "mlockall": true } } }
```

```
}
```

<1> The `max_file_descriptors` field shows the number of available descriptors that the Elasticsearch process can access.

Elasticsearch also uses a mix of NioFS and MMapFS (((("MMapFS"))))for the various files. Ensure that you configure the maximum map count so that there is ample virtual memory available for mmapped files. This can be set temporarily:

[source,js]

```
sysctl -w vm.max_map_count=262144
```



Or you can set it permanently by modifying `vm.max_map_count` setting in your `/etc/sysctl.conf`.



==== Revisit This List Before Production

You are likely reading this section before you go into production.

The details covered in this chapter are good to be generally aware of, but it is critical to revisit this entire list right before deploying to production.

Some of the topics will simply stop you cold (such as too few available file descriptors).

These are easy enough to debug because they are quickly apparent. Other issues, such as split brains and memory settings, are visible only after something bad happens. At that point, the resolution is often messy and tedious.

It is much better to proactively prevent these situations from occurring by configuring your cluster appropriately *before* disaster strikes. So if you are going to dog-ear (or bookmark) one section from the entire book, this chapter would be a good candidate. The week before deploying to production, simply flip through the list presented here and check off all the recommendations.







== Changing Settings Dynamically

Many settings in Elasticsearch are dynamic and can be modified through the API. Configuration changes that force a node (or cluster) restart are strenuously avoided.((("post-deployment", "changing settings dynamically")))) And while it's possible to make the changes through the static configs, we recommend that you use the API instead.

The `cluster-update` API operates(("Cluster Update API")) in two modes:

Transient:: These changes are in effect until the cluster restarts. Once a full cluster restart takes place, these settings are erased.

Persistent:: These changes are permanently in place unless explicitly changed. They will survive full cluster restarts and override the static configuration files.

Transient versus persistent settings are supplied in the JSON body:

[source,js]

```
PUT /_cluster/settings { "persistent" : { "discovery.zen.minimum_master_nodes" : 2 <1> },  
"transient" : { "indices.store.throttle.max_bytes_per_sec" : "50mb" <2> } }
```

```
}
```

<1> This persistent setting will survive full cluster restarts.

<2> This transient setting will be removed after the first full cluster restart.

A complete list of settings that can be updated dynamically can be found in the <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/cluster-update-settings.html>[online reference docs].



[[logging]] === Logging

Elasticsearch emits a number of logs, which are placed in `ES_HOME/logs`. The default logging level is `INFO`. (((("post-deployment", "logging"))))((("logging", "Elasticsearch logging"))) It provides a moderate amount of information, but is designed to be rather light so that your logs are not enormous.

When debugging problems, particularly problems with node discovery (since this often depends on finicky network configurations), it can be helpful to bump up the logging level to `DEBUG`.

You *could* modify the `logging.yml` file and restart your nodes--but that is both tedious and leads to unnecessary downtime. Instead, you can update logging levels through the `cluster-settings` API(((("Cluster Settings API, updating logging levels"))) that we just learned about.

To do so, take the logger you are interested in and prepend `logger.` to it. Let's turn up the discovery logging:

[source,js]

```
PUT /_cluster/settings { "transient" : { "logger.discovery" : "DEBUG" } }
```

```
}
```

While this setting is in effect, Elasticsearch will begin to emit `DEBUG`-level logs for the `discovery` module.

TIP: Avoid `TRACE`. It is extremely verbose, to the point where the logs are no longer useful.

[[slowlog]] ===== Slowlog

There is another log called the *slowlog*. The purpose of(((("Slowlog"))) this log is to catch queries and indexing requests that take over a certain threshold of time. It is useful for hunting down user-generated queries that are particularly slow.

By default, the slowlog is not enabled. It can be enabled by defining the action (query, fetch, or index), the level that you want the event logged at (`WARN`, `DEBUG`, and so forth) and a time threshold.

This is an index-level setting, which means it is applied to individual indices:



[source,js]

```
PUT /my_index/_settings { "index.search.slowlog.threshold.query.warn" : "10s", <1>
"index.search.slowlog.threshold.fetch.debug": "500ms", <2>
"index.indexing.slowlog.threshold.index.info": "5s" <3>

}
```

<1> Emit a `WARN` log when queries are slower than 10s.

<2> Emit a `DEBUG` log when fetches are slower than 500ms.

<3> Emit an `INFO` log when indexing takes longer than 5s.

You can also define these thresholds in your `elasticsearch.yml` file. Indices that do not have a threshold set will inherit whatever is configured in the static config.

Once the thresholds are set, you can toggle the logging level like any other logger:

[source,js]

```
PUT /_cluster/settings { "transient" : { "logger.index.search.slowlog" : "DEBUG", <1>
"logger.index.indexing.slowlog" : "WARN" <2> }

}
```

<1> Set the search slowlog to `DEBUG` level.

<2> Set the indexing slowlog to `WARN` level.



[[indexing-performance]] === Indexing Performance Tips

If you are in an indexing-heavy environment,((("indexing", "performance tips")))(("post-deployment", "indexing performance tips")) such as indexing infrastructure logs, you may be willing to sacrifice some search performance for faster indexing rates. In these scenarios, searches tend to be relatively rare and performed by people internal to your organization. They are willing to wait several seconds for a search, as opposed to a consumer facing a search that must return in milliseconds.

Because of this unique position, certain trade-offs can be made that will increase your indexing performance.

.These Tips Apply Only to Elasticsearch 1.3+

This book is written for the most recent versions of Elasticsearch, although much of the content works on older versions.

The tips presented in this section, however, are *explicitly* for version 1.3+. There have been multiple performance improvements and bugs fixed that directly impact indexing. In fact, some of these recommendations will *reduce* performance on older versions because of the presence of bugs or performance defects.

==== Test Performance Scientifically

Performance testing is always difficult, so try to be as scientific as possible in your approach. ((("performance testing")))(("indexing", "performance tips", "performance testing")) Randomly fiddling with knobs and turning on ingestion is not a good way to tune performance. If there are too many *causes*, it is impossible to determine which one had the best *effect*. A reasonable approach to testing is as follows:

1. Test performance on a single node, with a single shard and no replicas.
2. Record performance under 100% default settings so that you have a baseline to measure against.
3. Make sure performance tests run for a long time (30+ minutes) so you can evaluate long-term performance, not short-term spikes or latencies. Some events (such as segment merging, and GCs) won't happen right away, so the performance profile can change over time.
4. Begin making single changes to the baseline defaults. Test these rigorously, and if performance improvement is acceptable, keep the setting and move on to the next one.

==== Using and Sizing Bulk Requests



This should be fairly obvious, but use bulk indexing requests for optimal performance.

((("indexing", "performance tips", "bulk requests, using and sizing")))((("bulk API", "using and sizing bulk requests")))) Bulk sizing is dependent on your data, analysis, and cluster configuration, but a good starting point is 5–15 MB per bulk. Note that this is physical size. Document count is not a good metric for bulk size. For example, if you are indexing 1,000 documents per bulk, keep the following in mind:

- 1,000 documents at 1 KB each is 1 MB.
- 1,000 documents at 100 KB each is 100 MB.

Those are drastically different bulk sizes. Bulks need to be loaded into memory at the coordinating node, so it is the physical size of the bulk that is more important than the document count.

Start with a bulk size around 5–15 MB and slowly increase it until you do not see performance gains anymore. Then start increasing the concurrency of your bulk ingestion (multiple threads, and so forth).

Monitor your nodes with Marvel and/or tools such as `iostat`, `top`, and `ps` to see when resources start to bottleneck. If you start to receive `EsRejectedExecutionException`, your cluster can no longer keep up: at least one resource has reached capacity. Either reduce concurrency, provide more of the limited resource (such as switching from spinning disks to SSDs), or add more nodes.

[NOTE]

When ingesting data, make sure bulk requests are round-robined across all your data nodes. Do not send all requests to a single node, since that single node

will need to store all the bulks in memory while processing.

==== Storage

Disk are usually the bottleneck of any modern server. Elasticsearch heavily uses disks, and the more throughput your disks can handle, the more stable your nodes will be. Here are some tips for optimizing disk I/O:

- Use SSDs. As mentioned elsewhere, ((("storage")))((("indexing", "performance tips", "storage")))) they are superior to spinning media.



- Use RAID 0. Striped RAID will increase disk I/O, at the obvious expense of potential failure if a drive dies. Don't use mirrored or parity RAIDS since replicas provide that functionality.
- Alternatively, use multiple drives and allow Elasticsearch to stripe data across them via multiple `path.data` directories.
- Do not use remote-mounted storage, such as NFS or SMB/CIFS. The latency introduced here is antithetical to performance.
- If you are on EC2, beware of EBS. Even the SSD-backed EBS options are often slower than local instance storage.

[[segments-and-merging]] ===== Segments and Merging

Segment merging is computationally expensive,(((("indexing", "performance tips", "segments and merging"))))((("merging segments"))))((("segments", "merging"))) and can eat up a lot of disk I/O. Merges are scheduled to operate in the background because they can take a long time to finish, especially large segments. This is normally fine, because the rate of large segment merges is relatively rare.

But sometimes merging falls behind the ingestion rate. If this happens, Elasticsearch will automatically throttle indexing requests to a single thread. This prevents a *segment explosion* problem, in which hundreds of segments are generated before they can be merged. Elasticsearch will log `INFO`-level messages stating `now throttling indexing` when it detects merging falling behind indexing.

Elasticsearch defaults here are conservative: you don't want search performance to be impacted by background merging. But sometimes (especially on SSD, or logging scenarios), the throttle limit is too low.

The default is 20 MB/s, which is a good setting for spinning disks. If you have SSDs, you might consider increasing this to 100–200 MB/s. Test to see what works for your system:

[source,js]

```
PUT /_cluster/settings { "persistent" : { "indices.store.throttle.max_bytes_per_sec" : "100mb" }}
```

```
}
```

If you are doing a bulk import and don't care about search at all, you can disable merge throttling entirely. This will allow indexing to run as fast as your disks will allow:



[source,js]

```
PUT /_cluster/settings { "transient" : { "indices.store.throttle.type" : "none" <1> } }
```

```
}
```

<1> Setting the throttle type to `none` disables merge throttling entirely. When you are done importing, set it back to `merge` to reenable throttling.

If you are using spinning media instead of SSD, you need to add this to your `elasticsearch.yml`:

[source,yaml]

index.merge.scheduler.max_thread_count: 1

Spinning media has a harder time with concurrent I/O, so we need to decrease the number of threads that can concurrently access the disk per index. This setting will allow `max_thread_count + 2` threads to operate on the disk at one time, so a setting of `1` will allow three threads.

For SSDs, you can ignore this setting. The default is `Math.min(3, Runtime.getRuntime().availableProcessors() / 2)`, which works well for SSD.

Finally, you can increase `index.translog.flush_threshold_size` from the default 200 MB to something larger, such as 1 GB. This allows larger segments to accumulate in the translog before a flush occurs. By letting larger segments build, you flush less often, and the larger segments merge less often. All of this adds up to less disk I/O overhead and better indexing rates.

==== Other

Finally, there are some other considerations to keep in mind:

- If you don't need near real-time accuracy on your search results, consider dropping the `index.refresh_interval` of((("indexing", "performance tips", "other considerations")))((("refresh_interval setting")))) each index to `30s`. If you are doing a large import, you can disable refreshes by setting this value to `-1` for the duration of the import. Don't forget to reenable it when you are finished!



- If you are doing a large bulk import, consider disabling replicas by setting `index.number_of_replicas: 0 .(("replicas, disabling during large bulk imports"))` When documents are replicated, the entire document is sent to the replica node and the indexing process is repeated verbatim. This means each replica will perform the analysis, indexing, and potentially merging process. + In contrast, if you index with zero replicas and then enable replicas when ingestion is finished, the recovery process is essentially a byte-for-byte network transfer. This is much more efficient than duplicating the indexing process.
- If you don't have a natural ID for each document, use Elasticsearch's auto-ID functionality.((("id", "auto-ID functionality of Elasticsearch"))) It is optimized to avoid version lookups, since the autogenerated ID is unique.
- If you are using your own ID, try to pick an ID that is <http://bit.ly/1sDiR5t>[friendly to Lucene]. ((("UUIDs (universally unique identifiers")))) Examples include zero-padded sequential IDs, UUID-1, and nanotime; these IDs have consistent, sequential patterns that compress well. In contrast, IDs such as UUID-4 are essentially random, which offer poor compression and slow down Lucene.



[role="pagebreak-before"] === Rolling Restarts

There will come a time when you need to perform a rolling restart of your cluster--keeping the cluster online and operational, but taking nodes offline one at a time.((("rolling restart of your cluster")))((("clusters", "rolling restarts")))((("post-deployment", "rolling restarts"))))

The common reason is either an Elasticsearch version upgrade, or some kind of maintenance on the server itself (such as an OS update, or hardware). Whatever the case, there is a particular method to perform a rolling restart.

By nature, Elasticsearch wants your data to be fully replicated and evenly balanced. If you shut down a single node for maintenance, the cluster will immediately recognize the loss of a node and begin rebalancing. This can be irritating if you know the node maintenance is short term, since the rebalancing of very large shards can take some time (think of trying to replicate 1TB--even on fast networks this is nontrivial).

What we want to do is tell Elasticsearch to hold off on rebalancing, because we have more knowledge about the state of the cluster due to external factors. The procedure is as follows:

1. If possible, stop indexing new data. This is not always possible, but will help speed up recovery time.
2. Disable shard allocation. This prevents Elasticsearch from rebalancing missing shards until you tell it otherwise. If you know the maintenance window will be short, this is a good idea. You can disable allocation as follows: + [source,js]

```
PUT /_cluster/settings { "transient" : { "cluster.routing.allocation.enable" : "none" } }
```

}

1. Shut down a single node, preferably using the `shutdown` API on that particular machine: + [source,js]

POST /_cluster/nodes/_local/_shutdown

1. Perform a maintenance/upgrade.
2. Restart the node, and confirm that it joins the cluster.
3. Reenable shard allocation as follows: + [source,js]



```
PUT /_cluster/settings { "transient" : { "cluster.routing.allocation.enable" : "all" } }
```

}

+ Shard rebalancing may take some time. Wait until the cluster has returned to status green before continuing.

1. Repeat steps 2 through 6 for the rest of your nodes.
2. At this point you are safe to resume indexing (if you had previously stopped), but waiting until the cluster is fully balanced before resuming indexing will help to speed up the process.



[[backing-up-your-cluster]] === Backing Up Your Cluster

As with any software that stores data, it is important to routinely back up your data.

((("clusters", "Backing up")))((("post-deployment", "Backing up your cluster")))((("Backing up your cluster"))) Elasticsearch replicas provide high availability during runtime; they allow you to tolerate sporadic node loss without an interruption of service.

Replicas do not provide protection from catastrophic failure, however. For that, you need a real backup of your cluster--a complete copy in case something goes wrong.

To back up your cluster, you can use the `snapshot` API.((("snapshot-restore API"))) This will take the current state and data in your cluster and save it to a shared repository. This backup process is "smart." Your first snapshot will be a complete copy of data, but all subsequent snapshots will save the *delta* between the existing snapshots and the new data. Data is incrementally added and deleted as you snapshot data over time. This means subsequent backups will be substantially faster since they are transmitting far less data.

To use this functionality, you must first create a repository to save data. There are several repository types that you may choose from:

- Shared filesystem, such as a NAS
- Amazon S3
- HDFS (Hadoop Distributed File System)
- Azure Cloud

==== Creating the Repository

Let's set up a shared ((("Backing up your cluster", "Creating the repository")))((("filesystem repository"))) filesystem repository:

[source,js]

```
PUT _snapshot/my_backup <1> { "type": "fs", <2> "settings": { "location":  
"/mount/backups/my_backup" <3> }
```

```
}
```

<1> We provide a name for our repository, in this case it is called `my_backup`.

<2> We specify that the type of the repository should be a shared filesystem.

<3> And finally, we provide a mounted drive as the destination.

NOTE: The shared filesystem path must be accessible from all nodes in your cluster!



This will create the repository and required metadata at the mount point. There are also some other options that you may want to configure, depending on the performance profile of your nodes, network, and repository location:

```
max_snapshot_bytes_per_sec :: When snapshotting data into the repo, this controls the throttling of that process. The default is 20mb per second.
```

```
max_restore_bytes_per_sec :: When restoring data from the repo, this controls how much the restore is throttled so that your network is not saturated. The default is 20mb per second.
```

Let's assume we have a very fast network and are OK with extra traffic, so we can increase the defaults:

[source,js]

```
POST _snapshot/my_backup/ <1> { "type": "fs", "settings": { "location": "/mount/backups/my_backup", "max_snapshot_bytes_per_sec": "50mb", <2> "max_restore_bytes_per_sec": "50mb" }
```

```
}
```

<1> Note that we are using a `POST` instead of `PUT`. This will update the settings of the existing repository.

<2> Then add our new settings.

===== Snapshotting All Open Indices

A repository can contain multiple snapshots.((("indices", "open, snapshots on")))((("backing up your cluster", "snapshots on all open indexes"))) Each snapshot is associated with a certain set of indices (for example, all indices, some subset, or a single index). When creating a snapshot, you specify which indices you are interested in and give the snapshot a unique name.

Let's start with the most basic snapshot command:

[source,js]

PUT _snapshot/my_backup/snapshot_1



This will back up all open indices into a snapshot named `snapshot_1`, under the `my_backup` repository. This call will return immediately, and the snapshot will proceed in the background.

[TIP]

Usually you'll want your snapshots to proceed as a background process, but occasionally you may want to wait for completion in your script. This can be accomplished by adding a `wait_for_completion` flag:

[source,js]

PUT _snapshot/my_backup/snapshot_1? wait_for_completion=true

This will block the call until the snapshot has completed. Note that large snapshots may take a long time to return!

=====

==== Snapshotting Particular Indices

The default behavior is to back up all open indices.((("indices", "snapshotting particular")))((("backing up your cluster", "snapshotting particular indices"))) But say you are using Marvel, and don't really want to back up all the diagnostic `.marvel` indices. You just don't have enough space to back up everything.

In that case, you can specify which indices to back up when snapshotting your cluster:

[source,js]

```
PUT _snapshot/my_backup/snapshot_2 { "indices": "index_1,index_2"
```

```
}
```

This snapshot command will now back up only `index1` and `index2`.

==== Listing Information About Snapshots



Once you start accumulating snapshots in your repository, you may forget the details((("backing up your cluster", "listing information about snapshots"))) relating to each--particularly when the snapshots are named based on time demarcations (for example, `backup_2014_10_28`).

To obtain information about a single snapshot, simply issue a `GET` request against the repo and snapshot name:

[source,js]

GET _snapshot/my_backup/snapshot_2

This will return a small response with various pieces of information regarding the snapshot:

[source,js]

```
{ "snapshots": [ { "snapshot": "snapshot_1", "indices": [ ".marvel_2014_28_10", "index1", "index2" ], "state": "SUCCESS", "start_time": "2014-09-02T13:01:43.115Z", "start_time_in_millis": 1409662903115, "end_time": "2014-09-02T13:01:43.439Z", "end_time_in_millis": 1409662903439, "duration_in_millis": 324, "failures": [], "shards": { "total": 10, "failed": 0, "successful": 10 } } ] }
```

}

For a complete listing of all snapshots in a repository, use the `_all` placeholder instead of a snapshot name:

[source,js]

GET _snapshot/my_backup/_all

==== Deleting Snapshots

Finally, we need a command to delete old snapshots that ((("backing up your cluster", "deleting old snapshots"))are no longer useful. This is simply a `DELETE` HTTP call to the repo/snapshot name:



[source,js]

DELETE _snapshot/my_backup/snapshot_2

It is important to use the API to delete snapshots, and not some other mechanism (such as deleting by hand, or using automated cleanup tools on S3). Because snapshots are incremental, it is possible that many snapshots are relying on old segments. The `delete` API understands what data is still in use by more recent snapshots, and will delete only unused segments.

If you do a manual file delete, however, you are at risk of seriously corrupting your backups because you are deleting data that is still in use.

==== Monitoring Snapshot Progress

The `wait_for_completion` flag provides a rudimentary form of monitoring, but really isn't sufficient when snapshotting or restoring even moderately sized clusters.

Two other APIs will give you more-detailed status about the state of the snapshotting. First you can execute a `GET` to the snapshot ID, just as we did earlier get information about a particular snapshot:

[source,js]

GET _snapshot/my_backup/snapshot_3

If the snapshot is still in progress when you call this, you'll see information about when it was started, how long it has been running, and so forth. Note, however, that this API uses the same threadpool as the snapshot mechanism. If you are snapshotting very large shards, the time between status updates can be quite large, since the API is competing for the same threadpool resources.

A better option is to poll the `_status` API:

[source,js]

GET _snapshot/my_backup/snapshot_3/_status



The `_status` API returns immediately and gives a much more verbose output of statistics:

[source,js]

```
{ "snapshots": [ { "snapshot": "snapshot_3", "repository": "my_backup", "state": "IN_PROGRESS", <1> "shards_stats": { "initializing": 0, "started": 1, <2> "finalizing": 0, "done": 4, "failed": 0, "total": 5 }, "stats": { "number_of_files": 5, "processed_files": 5, "total_size_in_bytes": 1792, "processed_size_in_bytes": 1792, "start_time_in_millis": 1409663054859, "time_in_millis": 64 }, "indices": { "index_3": { "shards_stats": { "initializing": 0, "started": 0, "finalizing": 0, "done": 5, "failed": 0, "total": 5 }, "stats": { "number_of_files": 5, "processed_files": 5, "total_size_in_bytes": 1792, "processed_size_in_bytes": 1792, "start_time_in_millis": 1409663054859, "time_in_millis": 64 }, "shards": { "0": { "stage": "DONE", "stats": { "number_of_files": 1, "processed_files": 1, "total_size_in_bytes": 514, "processed_size_in_bytes": 514, "start_time_in_millis": 1409663054862, "time_in_millis": 22 } } } } } } } }
```

...

<1> A snapshot that is currently running will show `IN_PROGRESS` as its status.

<2> This particular snapshot has one shard still transferring (the other four have already completed).

The response includes the overall status of the snapshot, but also drills down into per-index and per-shard statistics. This gives you an incredibly detailed view of how the snapshot is progressing. Shards can be in various states of completion:

`INITIALIZING` :: The shard is checking with the cluster state to see whether it can be snapshotted. This is usually very fast.

`STARTED` :: Data is being transferred to the repository.

`FINALIZING` :: Data transfer is complete; the shard is now sending snapshot metadata.

`DONE` :: Snapshot complete!

`FAILED` :: An error was encountered during the snapshot process, and this shard/index/snapshot could not be completed. Check your logs for more information.

==== Canceling a Snapshot



Finally, you may want to cancel a snapshot or restore.(((“backing up your cluster”, “canceling a snapshot”))) Since these are long-running processes, a typo or mistake when executing the operation could take a long time to resolve--and use up valuable resources at the same time.

To cancel a snapshot, simply delete the snapshot while it is in progress:

[source,js]

DELETE _snapshot/my_backup/snapshot_3

This will halt the snapshot process. Then proceed to delete the half-completed snapshot from the repository.



==== Restoring from a Snapshot

Once you've backed up some data, restoring it is easy: simply add `_restore` to the ID of `((("post-deployment", "restoring from a snapshot"))((("restoring from a snapshot"))))` the snapshot you wish to restore into your cluster:

[source,js]

POST `_snapshot/my_backup/snapshot_1/_restore`

The default behavior is to restore all indices that exist in that snapshot. If `snapshot_1` contains five indices, all five will be restored into our cluster. `((("indices", "restoring from a snapshot")))` As with the `snapshot` API, it is possible to select which indices we want to restore.

There are also additional options for renaming indices. This allows you to match index names with a pattern, and then provide a new name during the restore process. This is useful if you want to restore old data to verify its contents, or perform some other processing, without replacing existing data. Let's restore a single index from the snapshot and provide a replacement name:

[source,js]

```
POST /snapshot/my_backup/snapshot_1/_restore { "indices": "index_1", <1>  
"rename_pattern": "index(.+)", <2> "renamereplacement": "restored_index$1" <3>
```

```
}
```

<1> Restore only the `index_1` index, ignoring the rest that are present in the snapshot.

<2> Find any indices being restored that match the provided pattern.

<3> Then rename them with the replacement pattern.

This will restore `index_1` into your cluster, but rename it to `restored_index_1`.

[TIP]



Similar to snapshotting, the `restore` command will return immediately, and the restoration process will happen in the background. If you would prefer your HTTP call to block until the restore is finished, simply add the `wait_for_completion` flag:

[source,js]

POST

`_snapshot/my_backup/snapshot_1/_restore?`
`wait_for_completion=true`

=====

==== Monitoring Restore Operations

The restoration of data from a repository piggybacks on the existing recovery mechanisms already in place in Elasticsearch.((("restoring from a snapshot", "monitoring restore operations"))) Internally, recovering shards from a repository is identical to recovering from another node.

If you wish to monitor the progress of a restore, you can use the `recovery` API. This is a general-purpose API that shows the status of shards moving around your cluster.

The API can be invoked for the specific indices that you are recovering:

[source,js]

GET /_recovery/restored_index_3

Or for all indices in your cluster, which may include other shards moving around, unrelated to your restore process:

[source,js]

GET /_recovery/

The output will look similar to this (and note, it can become very verbose depending on the activity of your cluster!):



[source,js]

```
{ "restored_index_3" : { "shards" : [ { "id" : 0, "type" : "snapshot", <1> "stage" : "index", "primary" : true, "start_time" : "2014-02-24T12:15:59.716", "stop_time" : 0, "total_time_in_millis" : 175576, "source" : { <2> "repository" : "my_backup", "snapshot" : "snapshot_3", "index" : "restored_index_3" }, "target" : { "id" : "ryqJ5lO5S4-ISFbGntkEkg", "hostname" : "my.fqdn", "ip" : "10.0.1.7", "name" : "my_es_node" }, "index" : { "files" : { "total" : 73, "reused" : 0, "recovered" : 69, "percent" : "94.5%" <3> }, "bytes" : { "total" : 79063092, "reused" : 0, "recovered" : 68891939, "percent" : "87.1%" }, "total_time_in_millis" : 0 }, "translog" : { "recovered" : 0, "total_time_in_millis" : 0 }, "start" : { "check_index_time" : 0, "total_time_in_millis" : 0 } } ] }
```

}

<1> The `type` field tells you the nature of the recovery; this shard is being recovered from a snapshot.

<2> The `source` hash describes the particular snapshot and repository that is being recovered from.

<3> The `percent` field gives you an idea about the status of the recovery. This particular shard has recovered 94% of the files so far; it is almost complete.

The output will list all indices currently undergoing a recovery, and then list all shards in each of those indices. Each shard will have stats about start/stop time, duration, recover percentage, bytes transferred, and more.

===== Canceling a Restore

To cancel a restore, you need to delete the indices being restored.((("restoring from a snapshot", "canceling a restore"))) Because a restore process is really just shard recovery, issuing a `delete-index` API alters the cluster state, which will in turn halt recovery. For example:

[source,js]

DELETE /restored_index_3

If `restored_index_3` was actively being restored, this delete command would halt the restoration as well as deleting any data that had already been restored into the cluster.



==== Clusters Are Living, Breathing Creatures

Once you get a cluster into production, you'll find that it takes on a life of its own. (((("clusters", "maintaining"))))((("post-deployment", "clusters, rolling restarts and upgrades"))))Elasticsearch works hard to make clusters self-sufficient and *just work*. But a cluster still requires routine care and feeding, such as routine backups and upgrades.

Elasticsearch releases new versions with bug fixes and performance enhancements at a very fast pace, and it is always a good idea to keep your cluster current. Similarly, Lucene continues to find new and exciting bugs in the JVM itself, which means you should always try to keep your JVM up-to-date.

This means it is a good idea to have a standardized, routine way to perform rolling restarts and upgrades in your cluster. Upgrading should be a routine process, rather than a once-yearly fiasco that requires countless hours of precise planning.

Similarly, it is important to have disaster recovery plans in place. Take frequent snapshots of your cluster--and periodically *test* those snapshots by performing a real recovery! It is all too common for organizations to make routine backups but never test their recovery strategy. Often you'll find a glaring deficiency the first time you perform a real recovery (such as users being unaware of which drive to mount). It's better to work these bugs out of your process with routine testing, rather than at 3 a.m. when there is a crisis.