

目录

Elasticsearch 权威指南

Introduction	1.1
序言	1.2
引言	1.3
谁应该读这这本书	1.3.1
为什么我们要写这本书	1.3.2
如何读这本书	1.3.3
本书导航	1.3.4
本书协议约定	1.3.5
基础入门	1.4
基础知识	1.4.1
安装运行elasticsearch	1.4.1.1
和Elasticsearch交互	1.4.1.2
面向文档	1.4.1.3
适应新环境	1.4.1.4
索引员工文档	1.4.1.5
检索文档	1.4.1.6
轻量检索	1.4.1.7
使用查询表达式搜索	1.4.1.8
更复杂的搜索	1.4.1.9
全文搜索	1.4.1.10
短语搜索	1.4.1.11
高亮搜索	1.4.1.12
分析	1.4.1.13
教程结语	1.4.1.14
分布式特性	1.4.1.15
集群内的原理	1.4.2
空集群	1.4.2.1
集群健康	1.4.2.2
添加索引	1.4.2.3
添加故障转移	1.4.2.4
水平扩容	1.4.2.5
应对故障	1.4.2.6
数据输入和输出	1.4.3

什么是文档	1.4.3.1
文档元数据	1.4.3.2
索引文档	1.4.3.3
取回一个文档	1.4.3.4
检查文档是否存在	1.4.3.5
更新整个文档	1.4.3.6
创建新文档	1.4.3.7
删除文档	1.4.3.8
处理冲突	1.4.3.9
乐观并发控制	1.4.3.10
文档的部分更新	1.4.3.11
取回多个文档	1.4.3.12
代价较小的批量操作	1.4.3.13
分布式文档存储	1.4.4
路由一个文档到一个分片中	1.4.4.1
主分片和副本分片如何交互	1.4.4.2
新建、索引和删除文档	1.4.4.3
取回一个文档	1.4.4.4
局部更新文档	1.4.4.5
多文档模式	1.4.4.6
搜索——最基本的工具	1.4.5
空搜索	1.4.5.1
多索引	1.4.5.2
分页	1.4.5.3
轻量 搜索	1.4.5.4
映射和分析	1.4.6
精确值 VS 全文	1.4.6.1
倒排索引	1.4.6.2
分析与分析器	1.4.6.3
映射	1.4.6.4
复杂核心域类型	1.4.6.5
请求体查询	1.4.7
空查询	1.4.7.1
查询表达式	1.4.7.2
查询与过滤	1.4.7.3
最重要的查询	1.4.7.4
组合多查询	1.4.7.5
验证查询	1.4.7.6

排序与相关性	1.4.8
排序	1.4.8.1
字符串排序与多字段	1.4.8.2
什么是相关性	1.4.8.3
聚合	1.5
高阶概念	1.5.1
桶	1.5.1.1
指标	1.5.1.2
尝试聚合	1.5.2
添加度量指标	1.5.2.1
嵌套桶	1.5.2.2
最后的修改	1.5.2.3
按时间统计	1.5.3
返回空 Buckets	1.5.3.1
扩展例子	1.5.3.2
潜力无穷	1.5.3.3
范围限定的聚合	1.5.4
过滤和聚合	1.5.5
过滤	1.5.5.1
过滤桶	1.5.5.2
后过滤器	1.5.5.3
小结	1.5.5.4
多桶排序	1.5.6
内置排序	1.5.6.1
按度量排序	1.5.6.2
基于“深度”度量排序	1.5.6.3
近似聚合	1.5.7
统计去重后的数量	1.5.7.1
百分位计算	1.5.7.2
通过聚合发现异常指标	1.5.8
统计去重后的数量	1.5.8.1
Doc Values and Fielddata	1.5.9
Doc Values	1.5.9.1
深入理解 Doc Values	1.5.9.2
聚合与分析	1.5.9.3
限制内存使用	1.5.9.4
Fielddata 的过滤	1.5.9.5
预加载 fielddata	1.5.9.6

优化聚合查询	1.5.9.7
总结	1.5.10
深入搜索	1.6
结构化搜索	1.6.1
精确值查找	1.6.1.1
控制相关度	1.6.2
相关度评分背后的理论	1.6.2.1
Lucene 的实用评分函数	1.6.2.2
查询时权重提升	1.6.2.3
使用查询结构修改相关度	1.6.2.4
Not Quite Not	1.6.2.5
忽略 TF/IDF	1.6.2.6
function_score 查询	1.6.2.7
按受欢迎度提升权重	1.6.2.8
过滤集提升权重	1.6.2.9
随机评分	1.6.2.10
越近越好	1.6.2.11
理解 price 价格语句	1.6.2.12
脚本评分	1.6.2.13
可插拔的相似度算法	1.6.2.14
更改相似度	1.6.2.15
调试相关度是最后 10% 要做的事情	1.6.2.16

- [1. GitBook](#)
- [2. 声明](#)

1. GitBook

Download PDF: [Elasticsearch:权威指南](#)

2. 声明

本书整理自官方文档

中文版本《[Elasticsearch:权威指南](#)》

英文版《[Elasticsearch: The Definitive Guide](#)》

[Elasticsearch官方最新文档](#)

[kibana使用手册官方中文文档](#)

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-07

- **1. 序言**

1. 序言

我仍然清晰地记得那个日子，我发布了这个开源项目第一个版本并在 IRC 聊天室创建一个频道，在那个最紧张的时刻，独自一人，急切地希望和盼望着第一个用户的到来。

第一个跳进 IRC 频道的用户就是 Clint（克林顿），当时我欣喜若狂。好吧…直到我发现 Clint 实际上是 Perl 用户啦，而且还是跟死亡讣告网站打交道。我记得（当时）问自己为什么他不是来自于更“主流”的社区，像 Ruby 或 Python，亦或是一个稍微好点的使用案例。

后来发生的一切都证明，我真是大错特错！Clint 最终对 Elasticsearch 的成功起到了重要作用。他是第一个将 Elasticsearch 投入生产环境的人（还是 0.4 的版本！），初期与 Clint 的交流和沟通对于将 Elasticsearch 塑造成今天的样子非常关键。对于什么是简单，Clint 有独特的见解并且他很少出错，这对 Elasticsearch 从管理、API 设计到日常使用等各个方面的易用性产生了深远的影响。所以公司成立不久，我们想也没想立即就联系 Clint，询问他是否愿意加入我们。

公司成立后，我们做的第一件事就是提供公开培训。很难表达我们当时有多么紧张和担心是否真的有人会报名。

但我们错了。

培训到现在依然很成功，很多主要城市都还有大量的人等待参加。参加培训的成员之中，有一个叫 Zach 年轻小伙吸引了我们注意。我们知道他写过很多关于 Elasticsearch 的博客（并暗自嫉妒他能够用非常简洁的方式来阐述复杂概念的能力），他还编写了一个 PHP 的客户端。然后我们发现 Zach 他还是自掏腰包来参加我们的培训！你真的不能要求更多，于是我们找到 Zach，问他是否愿意加入我们的公司。

Clint 和 Zach 是 Elasticsearch 能否成功的关键。他们是完美的解说家，从简单的上层应用到复杂的（Apache Lucene）底层逻辑。在 Elastic 这里我们非常珍惜这种独特技能。Clint 还负责 Elasticsearch Perl 客户端，而 Zach 则负责 PHP，都是精彩的代码。

最后，两位在 Elasticsearch 项目每天的日常事务中也扮演着重要的角色。Elasticsearch 如此受欢迎的主要原因之一，就是它拥有与用户沟通产生共鸣的能力，Clint 和 Zach 都是这个集体的一份子，这让一切成为可能。

Shay Banon

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- [1. 引言](#)

1. 引言

这个世界已然被数据淹没。多年来，我们系统间流转和产生的大量数据已让我们不知所措。现有的技术都集中在如何解决数据仓库存储以及如何结构化这些数据。这些看上去都挺美好，直到你实际需要基于这些数据实时做决策分析的时候才发现根本不是那么一回事。

Elasticsearch 是一个分布式、可扩展、实时的搜索与数据分析引擎。它能从项目一开始就赋予你的数据以搜索、分析和探索的能力，这是通常没有预料到的。它存在还因为原始数据如果只是躺在磁盘里面根本就毫无用处。

无论你是需要全文搜索，还是结构化数据的实时统计，或者两者结合，这本指南都能帮助你了解其中最基本的概念，从最基本的操作开始学习 Elasticsearch。之后，我们还会逐渐开始探索更加高级的搜索技术，不断提升搜索体验来满足你的需求。

Elasticsearch 不仅仅只是全文搜索，我们还将介绍结构化搜索、数据分析、复杂的人类语言处理、地理位置和对象间关联关系等。我们还将探讨为了充分利用 Elasticsearch 的水平伸缩性，应当如何建立数据模型，以及在生产环境中如何配置和监控你的集群。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- [1. 谁应该读这这本书](#)

1. 谁应该读这这本书

这本书是写给任何想要把他们的数据拿来干活做点事情的人。不管你是从头构建一个新项目，还是为了给已有的系统改造换血， Elasticsearch 都能够帮助你解决现有问题和开发新的功能，有些可能是你之前没有想到的功能。

这本书既适合初学者也适合有经验的用户。我们希望你有一定的编程基础，虽然不是必须的，但有用过 SQL 和关系数据库会更佳。我们会从原理解释和基本概念出发，帮助新手在复杂的搜索世界里打下稳固的知识基础。

具有搜索背景的读者也会受益于这本书。有经验的用户将懂得其所熟悉搜索的概念在 Elasticsearch 是如何对应和具体实现的。即使是高级用户，前面几个章节所包含的信息也是非常有用的。

最后，也许你是一名 DevOps，其他部门一直尽可能快的往 Elasticsearch 里面灌数据，而你是那个负责防止 Elasticsearch 服务器起火的消防员。只要用户在规则内行事，Elasticsearch 集群扩容相当轻松。不过你需要知道如何在进入生产环境前搭建一个稳定的集群，还能要在凌晨三点钟能识别出警告信号，以防止灾难发生。前面几章你可能不太感兴趣，但这本书的最后一部分是非常重要的，包含所有你需要知道的用以避免系统崩溃的知识。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- [1. 为什么要我们写这本书](#)

1. 为什么要我们写这本书

我们写这本书，因为 Elasticsearch 需要更好的阐述。现有的参考文档是优秀的一前提是你知道你在寻找什么。它假定你已经熟悉信息检索、分布式系统原理、Query DSL 和许多其他相关的概念。

这本书没有这样的假设。它的目的是写一本即便是什么都不懂的初学者（不管是对于搜索还是对于分布式系统）也能拿起它简单看完几章，就能开始搭建一个原型。

我们采取一种基于问题求解的方式：这是一个问题，我该怎么解决？如何对候选方案进行权衡取舍？我们从基础知识开始，循序渐进，每一章都建立在前一章之上，同时提供必要的实用案例和理论解释。

现有的参考文档解决了如何使用这些功能，我们希望这本书解决的是 **为什么** 和 **什么时候** 使用这些功能。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- 1. 如何读这本书

1. 如何读这本书

Elasticsearch 做了很多努力和尝试来让复杂的事情变得简单，很大程度上来说 Elasticsearch 的成功来源于此。换句话说，搜索以及分布式系统是非常复杂的，不过为了充分利用 Elasticsearch，迟早你也需要掌握它们。

恩，是有点复杂，但不是魔法。我们倾向于认为复杂系统如同神奇的黑盒子，能响应外部的咒语，但是通常里面的工作逻辑很简单。理解了这些逻辑过程你就能驱散魔法，理解内在能够让你更加明确和清晰，而不是寄托于黑盒子做你想要做的。

这本权威指南不仅会帮助你学习 Elasticsearch，而且希望能够带你接触一些更深入、更有趣的话题，如 集群内的原理、分布式文档存储、执行分布式检索 和 分片内部原理，这些虽然不是必要的阅读却能让你深入理解其内在机制。

本书的第一部分应该按章节顺序阅读，因为每一章建立在上一章的基础上（尽管你也可以浏览刚才提到的章节）。后续各章节如 近似匹配 和 部分匹配 相对独立，你可以按需选择性参阅。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- 1. 本书导航

1. 本书导航

这本书分为七个部分：

- 章节 你知道的, 为了搜索... 到 分片内部原理 主要是介绍 Elasticsearch。介绍了 Elasticsearch 的数据输入输出以及 Elasticsearch 如何处理你的文档数据。如何进行基本的搜索操作和管理你的索引。本章结束你将学会如何将 Elasticsearch 集成到你的应用程序中。章节：集群内的原理、分布式文档存储、执行分布式检索 和 分片内部原理 为附加章节，目的是让你了解分布式处理的过程，不是必读的。
- 章节 结构化搜索 到 控制相关度 带你深入了解搜索，如何借助一些更高级的特性，如邻近词（word proximity）和部分匹配（partial matching）来索引和查询你的数据。你将了解相关度评分是如何工作的以及如何控制它来确保第一页总是返回最佳的搜索结果。
- 章节 开始处理各种语言 到 拼写错误 解决如何有效使用分析器和查询来处理人类语言的棘手问题。我们会从一次简单的语言分析下手，然后逐步深入，如字母表和排序，还会涉及到词干提取、停用词、同义词和模糊匹配。
- 章节 高阶概念 到 Doc Values and Fielddata 讨论聚合（aggregations）和分析，对你的数据进行摘要化和分组来呈现总体趋势。
- 章节 地理坐标点 到 地理形状 介绍 Elasticsearch 支持的两种地理位置检索方式：经纬坐标点和复杂的地理形状（geo-shapes）。
- 章节 关联关系处理 到 扩容设计 谈到了为了高效使用 Elasticsearch，应当如何为你的数据建立模型。在搜索引擎里表达实体间的关系可能不是那么容易，因为它不是用来设计做这个的。这些章节还会阐述如何设计索引来匹配你系统中的数据流。
- 最后，章节 监控 到 部署后 将讨论生产环境上线的重要配置、监控点以及如何诊断以避免出现问题。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- **1. 本书协议约定**

1. 本书协议约定

以下是本书中使用的印刷规范：

斜体

表示重点、新的术语或概念。

等宽字体

用于程序列表以及在段落中引用变量或程序元素如：**函数名称、数据库、数据类型、环境变量、语句和关键字**。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- [1. 基础入门](#)

1. 基础入门

Elasticsearch 是一个实时的分布式搜索分析引擎，它能让你以前所未有的速度和规模，去探索你的数据。它被用作全文检索、结构化搜索、分析以及这三个功能的组合：

- Wikipedia 使用 Elasticsearch 提供带有高亮片段的全文搜索，还有 search-as-you-type 和 did-you-mean 的建议。
- 卫报 使用 Elasticsearch 将网络社交数据结合到访客日志中，为它的编辑们提供公众对于新文章的实时反馈。
- Stack Overflow 将地理位置查询融入全文检索中去，并且使用 more-like-this 接口去查找相关的问题和回答。
- GitHub 使用 Elasticsearch 对1300亿行代码进行查询。

Elasticsearch 不仅仅为巨头公司服务。它也帮助了很多初创公司，比如 Datadog 和 Klout，Elasticsearch 帮助他们将想法用原型实现，并转化为可扩展的解决方案。Elasticsearch 能运行在你的笔记本电脑上，或者扩展到数百台服务器上来处理PB级数据。

Elasticsearch 中没有一个单独的组件是全新的或者是革命性的。全文搜索很久之前就已经可以做到了，就像很早之前出现的分析系统和分布式数据库。革命性的成就是在将这些单独的、有用的组件融合到一个单一的、一致的、实时的应用中。对于初学者而言它的门槛相对较低，而当你的技能提升或需求增加时，它也始终能满足你的需求。

如果你现在打开这本书，是因为你拥有数据。除非你准备使用它 做些什么，否则拥有这些数据将没有意义。

不幸的是，大部分数据库在从你的数据中提取可用知识时出乎意料的低效。当然，你可以通过时间戳或精确值进行过滤，但是它们能够全文检索、处理同义词、通过相关性给文档评分么？它们能从同样的数据中生成分析与聚合数据吗？最重要的是，它们能实时地做到上述操作，而不经过大型批处理的任务么？

这就是 Elasticsearch 脱颖而出的地方：Elasticsearch 鼓励你去探索与利用数据，而不是因为查询数据太困难，就让它们烂在数据仓库里面。

Elasticsearch 将成为你最好的朋友。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 基础知识

1. 基础知识

Elasticsearch 是一个开源的搜索引擎，建立在一个全文搜索引擎库 Apache Lucene™ 基础之上。Lucene 可以说是当下最先进、高性能、全功能的搜索引擎库—无论是开源还是私有。

但是 Lucene 仅仅只是一个库。为了充分发挥其功能，你需要使用 Java 并将 Lucene 直接集成到应用程序中。更糟糕的是，您可能需要获得信息检索学位才能了解其工作原理。Lucene 非常 复杂。

Elasticsearch 也是使用 Java 编写的，它的内部使用 Lucene 做索引与搜索，但是它的目的是使全文检索变得简单，通过隐藏 Lucene 的复杂性，取而代之的提供一套简单一致的 RESTful API。

然而，Elasticsearch 不仅仅是 Lucene，并且也仅仅只是一个全文搜索引擎。它可以被下面这样准确的形容：

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

Elasticsearch 将所有的功能打包成一个单独的服务，这样你可以通过程序与它提供的简单的 RESTful API 进行通信，可以使用自己喜欢的编程语言充当 web 客户端，甚至可以使用命令行（去充当这个客户端）。

就 Elasticsearch 而言，起步很简单。对于初学者来说，它预设了一些适当的默认值，并隐藏了复杂的搜索理论知识。它 开箱即用。只需最少的理解，你很快就能具有生产力。

随着你知识的积累，你可以利用 Elasticsearch 更多的高级特性，它的整个引擎是可配置并且灵活的。从众多高级特性中，挑选恰当去修饰的 Elasticsearch，使它能解决你本地遇到的问题。

你可以免费下载，使用，修改 Elasticsearch。它在 [Apache 2 license](#) 协议下发布的，这是众多灵活的开源协议之一。Elasticsearch 的源码被托管在 Github 上 github.com/elastic/elasticsearch。如果你想加入我们这个令人惊奇的 contributors 社区，看这里 [Contributing to Elasticsearch](#)。

如果你对 Elasticsearch 有任何相关的问题，包括特定的特性(specific features)、语言客户端(language clients)、插件(plugins)，可以在discuss.elastic.co 加入讨论。

回忆时光

许多年前，一个刚结婚的名叫 Shay Banon 的失业开发者，跟着他的妻子去了伦敦，他的妻子在那里学习厨师。在寻找一个赚钱的工作的时候，为了给他的妻子做一个食谱搜索引擎，他开始使用 Lucene 的一个早期版本。

直接使用 Lucene 是很难的，因此 Shay 开始做一个抽象层，Java 开发者使用它可以很简单的给他们的程序添加搜索功能。他发布了他的第一个开源项目 Compass。

后来 Shay 获得了一份工作，主要是高性能，分布式环境下的内存数据网格。这个对于高性能，实时，分布式搜索引擎的需求尤为突出，他决定重写 Compass，把它变为一个独立的服务并取名 Elasticsearch。

第一个公开版本在2010年2月发布，从此以后，Elasticsearch 已经成为了 Github 上最活跃的项目之一，他拥有超过300名 contributors(目前736名 contributors)。一家公司已经开始围绕 Elasticsearch 提供商业服务，并开发新的特性，但是，Elasticsearch 将永远开源并对所有人可用。

据说，Shay 的妻子还在等着她的食谱搜索引擎...

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 安装运行elasticsearch

1. 安装运行elasticsearch

想用最简单的方式去理解 Elasticsearch 能为你做什么，那就是使用它了，让我们开始吧！

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-07

- [1. 和Elasticsearch交互](#)
 - [1.1. Java Api](#)
 - [1.2. Restful API with Json over HTTP](#)

1. 和Elasticsearch交互

和 Elasticsearch 的交互方式取决于你是否使用 Java。

1.1. Java Api

如果你正在使用 Java, 在代码中你可以使用 Elasticsearch 内置的两个客户端:

节点客户端 (Node client)

节点客户端作为一个非数据节点加入到本地集群中。换句话说, 它本身不保存任何数据, 但是它知道数据在集群中的哪个节点中, 并且可以把请求转发到正确的节点。

传输客户端 (Transport client)

轻量级的传输客户端可以将请求发送到远程集群。它本身不加入集群, 但是它可以将请求转发到集群中的一个节点上。

两个 Java 客户端都是通过 9300 端口并使用 Elasticsearch 的原生 传输 协议和集群交互。集群中的节点通过端口 9300 彼此通信。如果这个端口没有打开, 节点将无法形成一个集群。

建议

Java 客户端作为节点必须和 Elasticsearch 有相同的主要版本; 否则, 它们之间将无法互相理解。

更多的 Java 客户端信息可以在 [Elasticsearch Clients](#) 中找到。

1.2. Restful API with Json over HTTP

所有其他语言可以使用 RESTful API 通过端口 9200 和 Elasticsearch 进行通信, 你可以用你最喜爱的 web 客户端访问 Elasticsearch 。事实上, 正如你所看到的, 你甚至可以使用 curl 命令来和 Elasticsearch 交互。

注意

Elasticsearch 为以下语言提供了官方客户端—Groovy、JavaScript、.NET、PHP、Perl、Python 和 Ruby—还有很多社区提供的客户端和插件, 所有这些都可以在 [Elasticsearch Clients](#) 中找到。

一个 Elasticsearch 请求和任何 HTTP 请求一样由若干相同的部件组成:

```
curl -X<VERB> '<PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY_STRING>' -H 'Content-T'
```

被 < > 标记的部件：

变量	备注
VERB	适当的 HTTP 方法 或 谓词：GET、POST、PUT、HEAD 或者 DELETE。
PROTOCOL	http 或者 https (如果你在 Elasticsearch 前面有一个 https 代理)
HOST	Elasticsearch 集群中任意节点的主机名，或者用 localhost 代表本地机器上的节点。
PORT	运行 Elasticsearch HTTP 服务的端口号，默认是 9200。
PATH	API 的终端路径 (例如 _count 将返回集群中文档数量)。Path 可能包含多个组件，例如：_cluster/stats 和 _nodes/stats/jvm。
QUERY_STRING	任意可选的查询字符串参数 (例如 ?pretty 将格式化地输出 JSON 返回值，使其更容易阅读)
BODY	一个 JSON 格式的请求体 (如果请求需要的话)

例如，计算集群中文档的数量，我们可以用这个：

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

Elasticsearch 返回一个 HTTP 状态码 (例如：200 OK) 和 (除 HEAD 请求) 一个 JSON 格式的返回值。前面的 curl 请求将返回一个像下面一样的 JSON 体：

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

在返回结果中没有看到 HTTP 头信息是因为我们没有要求 curl 显示它们。想要看到头信息，需要结合 -i 参数来使用 curl 命令：

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

在书中剩余的部分，我们将用缩写格式来展示这些 curl 示例，所谓的缩写格式就是省略请求中所有相同的部分，例如主机名、端口号以及 curl 命令本身。而不是像下面显示的那样用一个完整的请求：

安装运行elasticsearch

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

我们将用缩写格式显示：

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

事实上， kibana DevTool 控制台 也使用这样相同的格式。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-06

- 1. 面向文档
- 2. JSON

1. 面向文档

在应用程序中对象很少只是一个简单的键和值的列表。通常，它们拥有更复杂的数据结构，可能包括日期、地理信息、其他对象或者数组等。

也许有一天你想把这些对象存储在数据库中。使用关系型数据库的行和列存储，这相当于把一个表现力丰富的对象塞到一个非常大的电子表格中：为了适应表结构，你必须设法将这个对象扁平化—通常一个字段对应一列—而且每次查询时又需要将其重新构造为对象。

Elasticsearch 是面向文档的，意味着它存储整个对象或文档。Elasticsearch 不仅存储文档，而且索引每个文档的内容，使之可以被检索。在 Elasticsearch 中，我们对文档进行索引、检索、排序和过滤—而不是对行列数据。这是一种完全不同的思考数据的方式，也是 Elasticsearch 能支持全文检索的原因。

2. JSON

Elasticsearch 使用 JavaScript Object Notation（或者 [JSON](#)）作为文档的序列化格式。JSON 序列化为大多数编程语言所支持，并且已经成为 NoSQL 领域的标准格式。它简单、简洁、易于阅读。

下面这个 JSON 文档代表了一个 user 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"  
}
```

虽然原始的 user 对象很复杂，但这个对象的结构和含义在 JSON 版本中都得到了体现和保留。在 Elasticsearch 中将对象转化为 JSON 后构建索引要比在一个扁平的表结构中要简单的多。

注意

几乎所有的语言都有可以将任意的数据结构或对象转化成 JSON 格式的模块，只是细节各不相同。具体请查看 `serialization` 或者 `marshalling` 这两个处理 JSON 的模块。官方 Elasticsearch 客户端自动为您提供 JSON 转化。

- [1. 适应新环境](#)
- [2. 创建一个雇员目录](#)

1. 适应新环境

为了让大家对 Elasticsearch 能实现什么及其上手难易程度有一个基本印象，让我们从一个简单的教程开始并介绍索引、搜索及聚合等基础概念。

我们将一并介绍一些新的技术术语，即使无法立即全部理解它们也无妨，因为在本书后续内容中，我们将继续深入介绍这里提到的所有概念。

接下来尽情享受 Elasticsearch 探索之旅。

2. 创建一个雇员目录

我们受雇于 Megacorp 公司，作为 HR 部门新的“热爱无人机”（"We love our drones!"）激励项目的一部分，我们的任务是为此创建一个员工目录。该目录应当能培养员工认同感及支持实时、高效、动态协作，因此有一些业务需求：

- 支持包含多值标签、数值、以及全文本的数据
- 检索任一员工的完整信息
- 允许结构化搜索，比如查询 30 岁以上的员工
- 允许简单的全文搜索以及较复杂的短语搜索
- 支持在匹配文档内容中高亮显示搜索片段
- 支持基于数据创建和管理分析仪表盘

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-07

- 1. 索引员工文档

1. 索引员工文档

第一个业务需求是存储员工数据。这将会以 员工文档 的形式存储：一个文档代表一个员工。存储数据到 Elasticsearch 的行为叫做 索引，但在索引一个文档之前，需要确定将文档存储在哪里。

一个 Elasticsearch 集群可以 包含多个 索引，相应的每个索引可以包含多个类型 (在6版本之后一个索引表只允许包含一个类型)。一个类型存储着多个 文档，每个文档又有 多个 属性。

Index Versus Index Versus Index

你也许已经注意到 索引 这个词在 Elasticsearch 语境中有多种含义，这里有必要做一些说明：

索引（名词）：

如前所述，一个 索引 类似于传统关系数据库中的一个 数据库，是一个存储关系型文档的地方。索引 (index) 的复数词为 indices 或 indexes。

索引（动词）：

索引一个文档 就是存储一个文档到一个 索引（名词） 中以便被检索和查询。这非常类似于 SQL 语句中的 INSERT 关键词，除了文档已存在时，新文档会替换旧文档情况之外。

倒排索引：

关系型数据库通过增加一个 索引 比如一个 B树 (B-tree) 索引 到指定的列上，以便提升数据检索速度。Elasticsearch 和 Lucene 使用了一个叫做 倒排索引 的结构来达到相同的目的。

默认的，一个文档中的每一个属性都是 被索引 的（有一个倒排索引）和可搜索的。一个没有倒排索引的属性是不能被搜索到的。我们将在 [倒排索引](#) 讨论倒排索引的更多细节。

对于员工目录，我们将做如下操作：

- 每个员工索引一个文档，文档包含该员工的所有信息。
- 每个文档都将是 *employee* 类型。
- 该类型位于 索引 *megacorp* 内。
- 该索引保存在我们的 Elasticsearch 集群中。

实践中这非常简单（尽管看起来有很多步骤），我们可以 通过一条命令 完成所有这些动作：

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

注意，路径 /megacorp/employee/1 包含了三部分的信息：

megacorp
索引名称
employee
类型名称
1
特定雇员的ID

请求体 —— JSON 文档 —— 包含了这位员工的所有详细信息，他的名字叫 John Smith，今年 25 岁，喜欢攀岩。

很简单！无需进行执行管理任务，如创建一个索引或指定每个属性的数据类型之类的，可以直接只索引一个文档。Elasticsearch 默认地完成其他一切，因此所有必需的管理任务都在后台使用默认设置完成。

进行下一步前，让我们增加更多的员工信息到目录中：

```
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" ]
}
```

Copyright © Songbai Yang all right reserved, powered by Gitbook 该文件修订时间：2021-03-07

- 1. 检索文档

1. 检索文档

目前我们已经在 Elasticsearch 中存储了一些数据，接下来就能专注于实现应用的业务需求了。第一个需求是可以检索到单个雇员的数据。

这在 Elasticsearch 中很简单。简单地执行一个 HTTP GET 请求并指定文档的地址——索引库、类型和ID。使用这三个信息可以返回原始的 JSON 文档：

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

返回结果包含了文档的一些元数据，以及 _source 属性，内容是 John Smith 雇员的原始 JSON 文档：

```
{
  "_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" ]
  }
}
```

建议

将 HTTP 命令由 PUT 改为 GET 可以用来检索文档，同样的，可以用 DELETE 命令来删除文档，以及使用 HEAD 指令来检查文档是否存在。如果想更新已存在的文档，只需再次 PUT 。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 轻量检索

1. 轻量检索

一个 GET 是相当简单的，可以直接得到指定的文档。现在尝试点儿稍微高级的功能，比如一个简单的搜索！

第一个尝试的几乎是最简单的搜索了。我们使用下列请求来搜索所有雇员：

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

可以看到，我们仍然使用索引库 megacorp 以及类型 employee，但与指定一个文档 ID 不同，这次使用 _search 。返回结果包括了所有三个文档，放在数组 hits 中。一个搜索默认返回十条结果。

```
{
  "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：返回结果不仅告知匹配了哪些文档，还包含了整个文档本身：显示搜索结果给最终用户所需的全部信息。

接下来，尝试下搜索姓氏为 `Smith` 的雇员。为此，我们将使用一个 高亮 搜索，很容易通过命令行完成。这个方法一般涉及到一个 查询字符串（query-string）搜索，因为我们通过一个URL参数来传递查询信息给搜索接口：

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

我们仍然在请求路径中使用 _search 端点，并将查询本身赋值给参数 q=。返回结果给出了所有的 Smith：

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 使用查询表达式搜索

1. 使用查询表达式搜索

Query-string 搜索通过命令非常方便地进行临时性的即席搜索，但它有自身的局限性（参见 [轻量搜索](#)）。Elasticsearch 提供一个丰富灵活的查询语言叫做 查询表达式，它支持构建更加复杂和健壮的查询。

领域特定语言（DSL），使用 JSON 构造了一个请求。我们可以像这样重写之前的查询所有名为 Smith 的搜索：

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

返回结果与之前的查询一样，但还是可以看到有一些变化。其中之一是，不再使用 query-string 参数，而是一个请求体替代。这个请求使用 JSON 构造，并使用了一个 match 查询（属于查询类型之一，后面将继续介绍）。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 更复杂的搜索

1. 更复杂的搜索

现在尝试下更复杂的搜索。同样搜索姓氏为 Smith 的员工，但这次我们只需要年龄大于 30 的。查询需要稍作调整，使用过滤器 filter，它支持高效地执行一个结构化查询。

```
GET /megacorp/employee/_search
{
  "query": {
    "bool": {
      "must": {
        "match": {
          "last_name": "smith" ( a)
        }
      },
      "filter": {
        "range": {
          "age": { "gt": 30 } ( b)
        }
      }
    }
  }
}
```

(a) 这部分与我们之前使用的 match 查询一样。

(b) 这部分是一个 range 过滤器，它能找到年龄大于 30 的文档，其中 gt 表示大于(great than)。

目前无需太多担心语法问题，后续会更详细地介绍。只需明确我们添加了一个过滤器用于执行一个范围查询，并复用之前的 match 查询。现在结果只返回了一名员工，叫 Jane Smith，32 岁。

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

- 1. 全文搜索

1. 全文搜索

截止目前的搜索相对都很简单：单个姓名，通过年龄过滤。现在尝试下稍微高级点儿的全文搜索——一项传统数据库确实很难搞定的任务。

搜索下所有喜欢攀岩（rock climbing）的员工：

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

显然我们依旧使用之前的 match 查询在 `about` 属性上搜索“rock climbing”。得到两个匹配的文档：

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

(a) 相关性得分

Elasticsearch 默认按照相关性得分排序，即每个文档跟查询的匹配程度。第一个最高得分的结果很明显：John Smith 的 `about` 属性清楚地写着“rock climbing”。

但为什么 Jane Smith 也作为结果返回了呢？原因是她的 about 属性里提到了“rock”。因为只有“rock”而没有“climbing”，所以她的相关性得分低于 John 的。

这是一个很好的案例，阐明了 Elasticsearch 如何在全文属性上搜索并返回相关性最强的结果。Elasticsearch 中的 相关性 概念非常重要，也是完全区别于传统关系型数据库的一个概念，数据库中的一条记录要么匹配要么不匹配。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-07

- 1. 短语搜索

1. 短语搜索

找出一个属性中的独立单词是没有问题的，但有时候想要精确匹配一系列单词或者短语。比如，我们想执行这样一个查询，仅匹配同时包含“rock”和“climbing”，并且二者以短语“rock climbing”的形式紧挨着的雇员记录。

为此对 match 查询稍作调整，使用一个叫做 match_phrase 的查询：

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

毫无悬念，返回结果仅有 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" ]
        }
      }
    ]
  }
}
```

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 高亮搜索

1. 高亮搜索

许多应用都倾向于在每个搜索结果中高亮部分文本片段，以便让用户知道为何该文档符合查询条件。在 Elasticsearch 中检索出高亮片段也很容易。

再次执行前面的查询，并增加一个新的 highlight 参数：

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

当执行该查询时，返回结果与之前一样，与此同时结果中还多了一个叫做 highlight 的部分。这个部分包含了 about 属性匹配的文本片段，并以 HTML 标签封装：

```
{
  ...
  "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>" ( a)
          ]
        }
      }
    ]
  }
}
```

(a) 原始文本中的高亮片段

关于高亮搜索片段，可以在 [highlighting reference documentation](#) 了解更多信息。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-06

- 1. 分析

1. 分析

终于到了最后一个业务需求：支持管理者对员工目录做分析。 Elasticsearch 有一个功能叫聚合（aggregations），允许我们基于数据生成一些精细的分析结果。聚合与 SQL 中的 GROUP BY 类似但更强大。

举个例子，挖掘出员工中最受欢迎的兴趣爱好：

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

暂时忽略掉语法，直接看看结果：

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

可以看到，两位员工对音乐感兴趣，一位对林业感兴趣，一位对运动感兴趣。这些聚合的结果数据并非预先统计，而是根据匹配当前查询的文档即时生成的。如果想知道叫 Smith 的员工中最受欢迎的兴趣爱好，可以直接构造一个组合查询：

安装运行elasticsearch

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

all_interests 聚合已经变为只包含匹配查询的文档：

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

聚合还支持分级汇总。比如，查询特定兴趣爱好员工的平均年龄：

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

得到的聚合结果有点儿复杂，但理解起来还是很简单的：

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

输出基本是第一次聚合的加强版。依然有一个兴趣及数量的列表，只不过每个兴趣都有了一个附加的 avg_age 属性，代表有这个兴趣爱好的所有员工的平均年龄。

即使现在不太理解这些语法也没有关系，依然很容易了解到复杂聚合及分组通过 Elasticsearch 特性实现得很完美，能够提取的数据类型也没有任何限制。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 教程结语

1. 教程结语

欣喜的是，这是一个关于 Elasticsearch 基础描述的教程，且仅仅是浅尝辄止，更多诸如 *suggestions*、*geolocation*、*percolation*、*fuzzy* 与 *partial matching* 等特性均被省略，以便保持教程的简洁。但它确实突显了开始构建高级搜索功能多么容易。不需要配置——只需要添加数据并开始搜索！

很可能语法会让你在某些地方有所困惑，并且对各个方面如何微调也有一些问题。没关系！本书后续内容将针对每个问题详细解释，让你全方位地理解 Elasticsearch 的工作原理。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- 1. 分布式特性

1. 分布式特性

在本章开头，我们提到过 Elasticsearch 可以横向扩展至数百（甚至数千）的服务器节点，同时可以处理PB级数据。我们的教程给出了一些使用 Elasticsearch 的示例，但并不涉及任何内部机制。Elasticsearch 天生就是分布式的，并且在设计时屏蔽了分布式的复杂性。

Elasticsearch 在分布式方面几乎是透明的。教程中并不要求了解分布式系统、分片、集群发现或其他的各种分布式概念。可以使用笔记本上的单节点轻松地运行教程里的程序，但如果你想要在 100 个节点的集群上运行程序，一切依然顺畅。

Elasticsearch 尽可能地屏蔽了分布式系统的复杂性。这里列举了一些在后台自动执行的操作：

- 分配文档到不同的容器 或 分片 中，文档可以储存在一个或多个节点中
- 按集群节点来均衡分配这些分片，从而对索引和搜索过程进行负载均衡
- 复制每个分片以支持数据冗余，从而防止硬件故障导致的数据丢失
- 将集群中任一节点的请求路由到存有相关数据的节点
- 集群扩容时无缝整合新节点，重新分配分片以便从离群节点恢复

当阅读本书时，将会遇到有关 Elasticsearch 分布式特性的补充章节。这些章节将介绍有关集群扩容、故障转移(集群内的原理)、应对文档存储(分布式文档存储)、执行分布式搜索(执行分布式检索)，以及分区 (shard) 及其工作原理(分片内部原理)。

这些章节并非必读，完全可以无需了解内部机制就使用 Elasticsearch，但是它们将从另一个角度帮助你了解更完整的 Elasticsearch 知识。可以根据需要跳过它们，或者想更完整地理解时再回头阅读也无妨。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 集群内的原理

1. 集群内的原理

补充章节

如前文所述，这是补充章节中第一篇介绍 Elasticsearch 在分布式环境中的运行原理。在这个章节中，我们将会介绍 cluster、node、shard 等常用术语，Elasticsearch 的扩容机制，以及如何处理硬件故障的内容。

虽然这个章节不是必读的—您完全可以在不关注分片、副本和失效切换等内容的情况下长期使用Elasticsearch-- 但是这将帮助你了解 Elasticsearch 的内部工作过程。您可以先快速阅览该章节，将来有需要时再次查看。

ElasticSearch 的主旨是随时可用和按需扩容。而扩容可以通过购买性能更强大（垂直扩容，或 纵向扩容）或者数量更多的服务器（水平扩容，或 横向扩容）来实现。

虽然 Elasticsearch 可以得益于更强大的硬件设备，但是垂直扩容是有极限的。真正的扩容能力是来自于水平扩容—为集群添加更多的节点，并且将负载压力和稳定性分散到这些节点中。

对于大多数的数据库而言，通常需要对应用程序进行非常大的改动，才能利用上横向扩容的新增资源。与之相反的是，ElastiSearch天生就是 分布式的，它知道如何通过管理多节点来提高扩容性和可用性。这也意味着你的应用无需关注这个问题。

本章将讲述如何按需配置集群、节点和分片，并在硬件故障时确保数据安全。

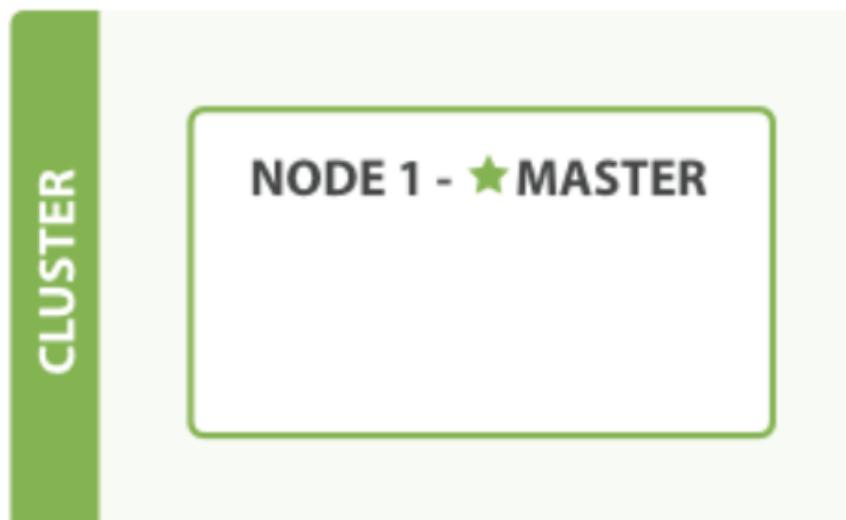
Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 空集群

1. 空集群

如果我们启动了一个单独的节点，里面不包含任何的数据和索引，那我们的集群看起来就是一个 Figure 1, “包含空内容节点的集群”。

Figure 1, “包含空内容节点的集群”



一个运行中的 Elasticsearch 实例称为一个节点，而集群是由一个或者多个拥有相同 `cluster.name` 配置的节点组成，它们共同承担数据和负载的压力。当有节点加入集群中或者从集群中移除节点时，集群将会重新平均分布所有的数据。

当一个节点被选举成为 主 节点时，它将负责管理集群范围内的所有变更，例如增加、删除索引，或者增加、删除节点等。而主节点并不需要涉及到文档级别的变更和搜索等操作，所以当集群只拥有一个主节点的情况下，即使流量的增加它也不会成为瓶颈。任何节点都可以成为主节点。我们的示例集群就只有一个节点，所以它同时也成为了主节点。

作为用户，我们可以将请求发送到 集群中的任何节点，包括主节点。每个节点都知道任意文档所处的位置，并且能够将我们的请求直接转发到存储我们所需文档的节点。无论我们将请求发送到哪个节点，它都能负责从各个包含我们所需文档的节点收集回数据，并将最终结果返回给客户端。Elasticsearch 对这一切的管理都是透明的。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- 1. 集群健康

1. 集群健康

Elasticsearch 的集群监控信息中包含了许多的统计数据，其中最为重要的一项就是 集群健康， 它在 status 字段中展示为 green 、 yellow 或者 red 。

```
GET /_cluster/health
```

在一个不包含任何索引的空集群中，它将会有一个类似于如下所示的返回内容：

```
{  
  "cluster_name": "elasticsearch",  
  "status": "yellow", (a)  
  "timed_out": false,  
  "number_of_nodes": 10,  
  "number_of_data_nodes": 9,  
  "active_primary_shards": 6921,  
  "active_shards": 13661,  
  "relocating_shards": 0,  
  "initializing_shards": 0,  
  "unassigned_shards": 181,  
  "delayed_unassigned_shards": 0,  
  "number_of_pending_tasks": 0,  
  "number_of_in_flight_fetch": 0,  
  "task_max_waiting_in_queue_millis": 0,  
  "active_shards_percent_as_number": 98.6923854934258  
}
```

(a)status 字段是我们最关心的。

status 字段指示着当前集群在总体上是否工作正常。它的三种颜色含义如下：

green

所有的主分片和副本分片都正常运行。

yellow

所有的主分片都正常运行，但不是所有的副本分片都正常运行。

red

有主分片没能正常运行。

在本章节剩余的部分，我们将解释什么是 主 分片和 副本 分片，以及上面提到的这些颜色的实际意义。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 添加索引

1. 添加索引

我们往 Elasticsearch 添加数据时需要用到 索引 —— 保存相关数据的地方。索引实际上是指向一个或者多个物理 分片 的 逻辑命名空间。

一个 分片 是一个底层的工作单元，它仅保存了全部数据中的一部分。在分片内部机制中，我们将详细介绍分片是如何工作的，而现在我们只需知道一个分片是一个 Lucene 的实例，以及它本身就是一个完整的搜索引擎。我们的文档被存储和索引到分片内，但是应用程序是直接与索引而不是与分片进行交互。

Elasticsearch 是利用分片将数据分发到集群内各处的。分片是数据的容器，文档保存在分片内，分片又被分配到集群内的各个节点里。当你的集群规模扩大或者缩小时，Elasticsearch 会自动的在各节点中迁移分片，使得数据仍然均匀分布在集群里。

一个分片可以是 主 分片或者 副本 分片。索引内任意一个文档都归属于一个主分片，所以主分片的数目决定着索引能够保存的最大数据量。

注意

技术上来说，一个主分片最大能够存储 Integer.MAX_VALUE - 128 个文档，但是实际最大值还需要参考你的使用场景：包括你使用的硬件，文档的大小和复杂程度，索引和查询文档的方式以及你期望的响应时长。

一个副本分片只是一个主分片的拷贝。副本分片作为硬件故障时保护数据不丢失的冗余备份，并为搜索和返回文档等读操作提供服务。

在索引建立的时候就已经确定了主分片数，但是副本分片数可以随时修改。

让我们在包含一个空节点的集群内创建名为 blogs 的索引。索引在默认情况下会被分配5个主分片，但是为了演示目的，我们将分配3个主分片和一份副本（每个主分片拥有一个副本分片）：

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

我们的集群现在是Figure 2，“拥有一个索引的单节点集群”。所有3个主分片都被分配在 Node 1。

Figure 2. 拥有一个索引的单节点集群



如果我们现在查看[集群健康](#), 我们将看到如下内容:

```
{  
  "cluster_name": "elasticsearch",  
  "status": "yellow", (a)  
  "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, (b)  
  "delayed_unassigned_shards": 0,  
  "number_of_pending_tasks": 0,  
  "number_of_in_flight_fetch": 0,  
  "task_max_waiting_in_queue_millis": 0,  
  "active_shards_percent_as_number": 50  
}
```

- (a)集群 *status* 值为 *yellow*。
- (b)没有被分配到任何节点的副本数。

集群的健康状况为 *yellow* 则表示全部 主 分片都正常运行（集群可以正常服务所有请求），但是 副本 分片没有全部处在正常状态。实际上，所有3个副本分片都是 *unassigned* —— 它们都没有被分配到任何节点。在同一个节点上既保存原始数据又保存副本是没有意义的，因为一旦失去了那个节点，我们也将丢失该节点上的所有副本数据。

当前我们的集群是正常运行的，但是在硬件故障时有丢失数据的风险。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间: 2021-03-06

- 1. 添加故障转移

1. 添加故障转移

当集群中只有一个节点在运行时，意味着会有一个单点故障问题——没有冗余。幸运的是，我们只需再启动一个节点即可防止数据丢失。

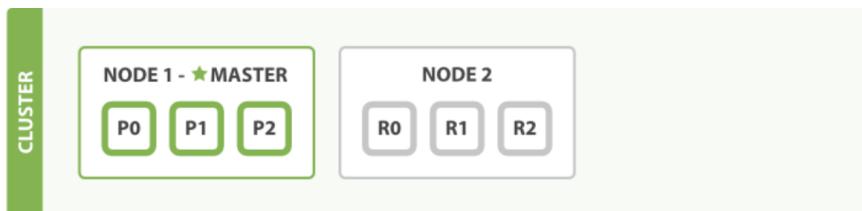
启动第二个节点

为了测试第二个节点启动后的情况，你可以在同一个目录内，完全依照启动第一个节点的方式来启动一个新节点（参考安装并运行 Elasticsearch）。多个节点可以共享同一个目录。

当你在同一台机器上启动了第二个节点时，只要它和第一个节点有同样的 cluster.name 配置，它就会自动发现集群并加入到其中。但是在不同机器上启动节点的时候，为了加入到同一集群，你需要配置一个可连接到的单播主机列表。详细信息请查看最好使用单播代替组播

如果启动了第二个节点，我们的集群将会如Figure 3，“拥有两个节点的集群——所有主分片和副本分片都已被分配”所示。

Figure 3. 拥有两个节点的集群——所有主分片和副本分片都已被分配



当第二个节点加入到集群后，3个 副本分片 将会分配到这个节点上——每个主分片对应一个副本分片。这意味着当集群内任何一个节点出现问题时，我们的数据都完好无损。

所有新近被索引的文档都将会保存在主分片上，然后被并行的复制到对应的副本分片上。这就保证了我们既可以从主分片又可以从副本分片上获得文档。

`cluster-health` 现在展示的状态为 `green`，这表示所有6个分片（包括3个主分片和3个副本分片）都在正常运行。

```
{  
  "cluster_name": "elasticsearch",  
  "status": "green", (a)  
  "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,  
  "delayed_unassigned_shards": 0,  
  "number_of_pending_tasks": 0,  
  "number_of_in_flight_fetch": 0,  
  "task_max_waiting_in_queue_millis": 0,  
  "active_shards_percent_as_number": 100  
}
```

(a)集群 status 值为 green。

我们的集群现在不仅仅是正常运行的，并且还处于始终可用的状态。

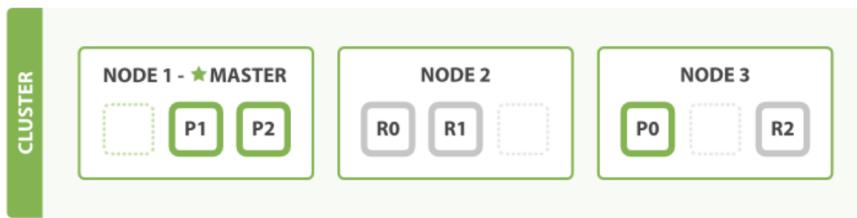
Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-06

- 1. 水平扩容
- 2. 更多的扩容

1. 水平扩容

怎样为我们的正在增长中的应用程序按需扩容呢？当启动了第三个节点，我们的集群将会看起来如Figure 4，“拥有三个节点的集群——为了分散负载而对分片进行重新分配”所示。

Figure 4. 拥有三个节点的集群——为了分散负载而对分片进行重新分配



Node 1 和 Node 2 上各有一个分片被迁移到了新的 Node 3 节点，现在每个节点上都拥有2个分片，而不是之前的3个。这表示每个节点的硬件资源（CPU, RAM, I/O）将被更少的分片所共享，每个分片的性能将会得到提升。

分片是一个功能完整的搜索引擎，它拥有使用一个节点上的所有资源的能力。我们这个拥有6个分片（3个主分片和3个副本分片）的索引可以最大扩容到6个节点，每个节点上存在一个分片，并且每个分片拥有所在节点的全部资源。

2. 更多的扩容

但是如果我们要扩容超过6个节点怎么办呢？

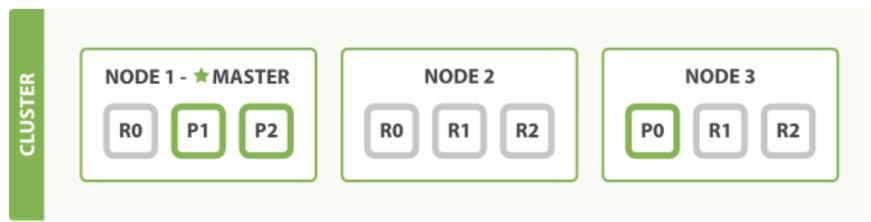
主分片的数目在索引创建时就已经确定了下来。实际上，这个数目定义了这个索引能够存储的最大数据量。（实际大小取决于你的数据、硬件和使用场景。）但是，读操作——搜索和返回数据——可以同时被主分片或副本分片所处理，所以当你拥有越多的副本分片时，也将拥有越高的吞吐量。

在运行中的集群上是可以动态调整副本分片数目的，我们可以按需伸缩集群。让我们把副本数从默认的 1 增加到 2：

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

如Figure 5，“将参数 number_of_replicas 调大到 2”所示， blogs 索引现在拥有9个分片：3个主分片和6个副本分片。这意味着我们可以将集群扩容到9个节点，每个节点上一个分片。相比原来3个节点时，集群搜索性能可以提升 3 倍。

Figure 5. 将参数 number_of_replicas 调大到 2



注意

当然，如果只是在相同节点数目的集群上增加更多的副本分片并不能提高性能，因为每个分片从节点上获得的资源会变少。你需要增加更多的硬件资源来提升吞吐量。但是更多的副本分片数提高了数据冗余量：按照上面的节点配置，我们可以在失去2个节点的情况下不丢失任何数据。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-06

- 1. 应对故障

1. 应对故障

我们之前说过 Elasticsearch 可以应对节点故障，接下来让我们尝试下这个功能。如果我们关闭第一个节点，这时集群的状态为Figure 6, “关闭了一个节点后的集群”

Figure 6. 关闭了一个节点后的集群



我们关闭的节点是一个主节点。而集群必须拥有一个主节点来保证正常工作，所以发生的第一件事情就是选举一个新的主节点：Node 2。

在我们关闭 Node 1 的同时也失去了主分片 1 和 2，并且在缺失主分片的时候索引也不能正常工作。如果此时来检查集群的状况，我们看到的状态将会为 red：不是所有主分片都在正常工作。

幸运的是，在其它节点上存在着这两个主分片的完整副本，所以新的主节点立即将这些分片在 Node 2 和 Node 3 上对应的副本分片提升为主分片，此时集群的状态将会为 yellow。这个提升主分片的过程是瞬间发生的，如同按下一个开关一般。

为什么我们集群状态是 yellow 而不是 green 呢？虽然我们拥有了所有的三个主分片，但是同时设置了每个主分片需要对应2份副本分片，而此时只存在一份副本分片。所以集群不能为 green 的状态，不过我们不必过于担心：如果我们同样关闭了 Node 2，我们的程序依然可以保持在不丢任何数据的情况下运行，因为 Node 3 为每一个分片都保留着一份副本。

如果我们重新启动 Node 1，集群可以将缺失的副本分片再次进行分配，那么集群的状态也将如Figure 5, “将参数 number_of_replicas 调大到 2”所示。如果 Node 1 依然拥有着之前的分片，它将尝试去重用它们，同时仅从主分片复制发生了修改的数据文件。

到目前为止，你应该对分片如何使得 Elasticsearch 进行水平扩容以及数据保障等知识有了一定了解。接下来我们将讲述关于分片生命周期的更多细节。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- 1. 数据输入和输出

1. 数据输入和输出

无论我们写什么样的程序，目的都是一样的：以某种方式组织数据服务我们的目的。但是数据不仅仅由随机位和字节组成。我们建立数据元素之间的关系以便于表示实体，或者现实世界中存在的事物。如果我们知道一个名字和电子邮件地址属于同一个人，那么它们将会更有意义。

尽管在现实世界中，不是所有的类型相同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，而另一个人只有一个手机号码，再一个人可能两者兼有。一个人可能有三个电子邮件地址，而另一个人却一个都没有。一位西班牙人可能有两个姓，而讲英语的人可能只有一个姓。

面向对象编程语言如此流行的原因之一是对象帮我们表示和处理现实世界具有潜在的复杂的数据结构的实体，到目前为止，一切都很完美！

但是当我们需要存储这些实体时问题来了，传统上，我们以行和列的形式存储数据到关系型数据库中，相当于使用电子表格。正因为我们使用了这种不灵活的存储媒介导致所有我们使用对象的灵活性都丢失了。

但是是否我们可以将我们的对象按对象的方式来存储？这样我们就能更加专注于使用数据，而不是在电子表格的局限性下对我们的应用建模。我们可以重新利用对象的灵活性。

一个对象是基于特定语言的内存的数据结构。为了通过网络发送或者存储它，我们需要将它表示成某种标准的格式。JSON是一种以人可读的文本表示对象的方法。它已经变成NoSQL世界交换数据的事实标准。当一个对象被序列化成为JSON，它被称为一个JSON文档。

Elasticsearch是分布式的文档存储。它能存储和检索复杂的数据结构—序列化成为JSON文档—以实时的方式。换句话说，一旦一个文档被存储在Elasticsearch中，它就是可以被集群中的任意节点检索到。

当然，我们不仅要存储数据，我们一定还需要查询它，成批且快速的查询它们。尽管现存的NoSQL解决方案允许我们以文档的形式存储对象，但是他们仍旧需要我们思考如何查询我们的数据，以及确定哪些字段需要被索引以加快数据检索。

在Elasticsearch中，每个字段的所有数据都是默认被索引的。即每个字段都有为了快速检索设置的专用倒排索引。而且，不像其他多数的数据库，它能在同一个查询中使用所有这些倒排索引，并以惊人的速度返回结果。

在本章中，我们展示了用来创建，检索，更新和删除文档的API。就目前而言，我们不关心文档中的数据或者怎样查询它们。所有我们关心的就是在Elasticsearch中怎样安全的存储文档，以及如何将文档再次返回。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-06

- 1. 什么是文档

1. 什么是文档

在大多数应用中，多数实体或对象可以被序列化为包含键值对的 JSON 对象。一个键可以是一个字段或字段的名称，一个值可以是一个字符串，一个数字，一个布尔值，另一个对象，一些数组值，或一些其它特殊类型诸如表示日期的字符串，或代表一个地理位置的对象：

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

通常情况下，我们使用的术语 **对象** 和 **文档** 是可以互相替换的。不过，有一个区别：一个对象仅仅是类似于 hash、hashmap、字典或者关联数组的 JSON 对象，对象中也可以嵌套其他的对象。对象可能包含了另外一些对象。在 Elasticsearch 中，术语 **文档** 有着特定的含义。它是指最顶层或者根对象，这个根对象被序列化成 JSON 并存储到 Elasticsearch 中，指定了唯一 ID。

⚠ 警示

字段的名字可以是任何合法的字符串，但不可以包含英文句号(.)。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-07

- **1. 文档元数据**
 - **1.1. _index**
 - **1.2. _type**
 - **1.3. _id**
 - **1.4. 其他元数据**

1. 文档元数据

一个文档不仅仅包含它的数据，也包含元数据——有关文档的信息。三个必须的元数据元素如下：

`_index`
文档在哪存放

`_type`
文档表示的对象类别

`_id`
文档唯一标识

1.1. `_index`

一个索引应该是因共同的特性被分组到一起的文档集合。例如，你可能存储所有的产品在索引 `products` 中，而存储所有销售的交易到索引 `sales` 中。虽然也允许存储不相关的数据到一个索引中(6.x版本之前可以这么做)，但这通常看作是一个反模式的做法。

建议

实际上，在 Elasticsearch 中，我们的数据是被存储和索引在分片中，而一个索引仅仅是逻辑上的命名空间，这个命名空间由一个或者多个分片组合在一起。然而，这是一个内部细节，我们的应用程序根本不应该关心分片，对于应用程序而言，只需知道文档位于一个索引内。Elasticsearch 会处理所有的细节。

我们将在 索引管理 介绍如何自行创建和管理索引，但现在我们将让 Elasticsearch 帮我们创建索引。所有需要我们做的就是选择一个索引名，这个名字必须小写，不能以下划线开头，不能包含逗号。我们用 `website` 作为索引名举例。

1.2. `_type`

数据可能在索引中只是松散的组合在一起，但是通常明确定义一些数据中的子分区是有用的。例如，所有的产品都放在一个索引中，但是你有许多不同的产品类别，比如 `"electronics"`、`"kitchen"` 和 `"lawn-care"`。这些文档共享一种相同的（或非常相似）的模式：他们有一个标题、描述、产品代码和价格。他们只是正好属于“产品”下的一些子类。Elasticsearch 公开了一个称为 `types`（类型）的特性，它允许您在索引中对数据进行逻辑分区。不同 `types` 的文档可能有不同的字段，但最好能够非常相似。我们将在 类型和映射 中更多的讨论关于 `types` 的一些应用和限制。一个 `_type` 命名可以是大写或者小写，但是不能以下划线或者句号开头，不应该包含逗号，并且长度限制为256个字符。我们使用 `blog` 作为类型名举例。

⚠ 警示

在 ES6.x 之后已经不支持单个索引表多个 type， type 已经没有任何实际意义了，在 Elasticsearch7.x 中 type 默认值为 doc

1.3. _id

ID 是一个字符串，当它和 _index 以及 _type 组合就可以唯一确定 Elasticsearch 中的一个文档。当你创建一个新的文档，要么提供自己的 _id，要么让 Elasticsearch 帮你生成。

1.4. 其他元数据

还有一些其他的元数据元素，他们在 [类型和映射](#) 进行了介绍。通过前面已经列出的元数据元素，我们已经能存储文档到 Elasticsearch 中并通过 ID 检索它—换句话说，使用 Elasticsearch 作为文档的存储介质。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-07

- [1. 索引文档](#)
 - [1.1. 使用自定义的ID](#)
 - [1.2. 自动生成id](#)

1. 紴索引文档

通过使用 `index API`，文档可以被索引——存储和使文档可被搜索。但是首先，我们要确定文档的位置。正如我们刚刚讨论的，一个文档的 `_index`、`_type` 和 `_id` 唯一标识一个文档。我们可以提供自定义的 `_id` 值，或者让 `index API` 自动生成。

1.1. 使用自定义的ID

如果你的文档有一个自然的标识符（例如，一个 `user_account` 字段或其他标识文档的值），你应该使用如下方式的 `index API` 并提供你自己 `_id`：

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

举个例子，如果我们的索引称为 `website`，类型称为 `blog`，并且选择 `123` 作为 `ID`，那么索引请求应该是下面这样：

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

Elasticsearch 响应体如下所示：

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

该响应表明文档已经成功创建，该索引包括 `_index`、`_type` 和 `_id` 元数据，以及一个新元素：`_version`。

在 Elasticsearch 中每个文档都有一个版本号。当每次对文档进行修改时（包括删除），`_version` 的值会递增。在处理冲突中，我们讨论了怎样使用 `_version` 号码确保你的应用程序中的一部分修改不会覆盖另一部分所做的修改。

1.2. 自动生成id

如果你的数据没有自然的 ID， Elasticsearch 可以帮我们自动生成 ID。请求的结构调整为：不再使用 PUT 谓词(“使用这个 URL 存储这个文档”), 而是使用 POST 谓词(“存储文档在这个 URL 命名空间下”)。

现在该 URL 只需包含 _index 和 _type :

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

除了 _id 是 Elasticsearch 自动生成的，响应的其他部分和前面的类似：

```
{
  "_index": "website",
  "_type": "blog",
  "_id": "AVFgSgVHUP18jI2wRx0w",
  "_version": 1,
  "created": true
}
```

自动生成的 ID 是 URL-safe、基于 Base64 编码且长度为20个字符的 GUID 字符串。这些 GUID 字符串由可修改的 FlakelID 模式生成，这种模式允许多个节点并行生成唯一 ID，且互相之间的冲突概率几乎为零。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-07

- [1. 取回一个文档](#)
- [2. 返回文档的一部分](#)

1. 取回一个文档

为了从 Elasticsearch 中检索出文档，我们仍然使用相同的 `_index` , `_type` , 和 `_id` , 但是 `HTTP` 谓词更改为 `GET` :

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

响应体包括目前已经熟悉了的元数据元素，再加上 `_source` 字段，这个字段包含我们索引数据时发送给 Elasticsearch 的原始 JSON 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"
  }
}
```

注意

在请求的查询串参数中加上 `pretty` 参数，正如前面的例子中看到的，这将会调用 Elasticsearch 的 pretty-print 功能，该功能使得 JSON 响应体更加可读。但是，`_source` 字段不能被格式化打印出来。相反，我们得到的 `_source` 字段中的 JSON 串，刚好是和我们传给它的一样。

`GET` 请求的响应体包括 `{"found": true}`，这证实了文档已经被找到。如果我们请求一个不存在的文档，我们仍旧会得到一个 JSON 响应体，但是 `found` 将会是 `false`。此外，`HTTP` 响应码将会是 `404 Not Found`，而不是 `200 OK`。

我们可以通过传递 `-i` 参数给 `curl` 命令，该参数能够显示响应的头部：

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

显示响应头部的响应体现在类似这样：

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

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

2. 返回文档的一部分

默认情况下， GET 请求会返回整个文档，这个文档正如存储在 _source 字段中的一样。但是也许你只对其中的 title 字段感兴趣。单个字段能用 _source 参数请求得到，多个字段也能使用逗号分隔的列表来指定。

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

该 _source 字段现在包含的只是我们请求的那些字段，并且已经将 date 字段过滤掉了。

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

或者，如果你只想得到 _source 字段，不需要任何元数据，你能使用 _source 端点：

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

那么返回的内容如下所示：

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

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-03-07

- 1. 检查文档是否存在

1. 检查文档是否存在

如果只想检查一个文档是否存在--根本不想关心内容--那么用 HEAD 方法来代替 GET 方法。 HEAD 请求没有返回体, 只返回一个 HTTP 请求报头:

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

如果文档存在, Elasticsearch 将返回一个 200 ok 的状态码:

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

若文档不存在, Elasticsearch 将返回一个 404 Not Found 的状态码:

```
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, 一个文档仅仅是在检查的时候不存在, 并不意味着一毫秒之后它也不存在: 也许同时正好另一个进程就创建了该文档。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间: 2021-03-08

- 1. 更新整个文档

1. 更新整个文档

在 Elasticsearch 中文档是不可改变的，不能修改它们。相反，如果想要更新现有的文档，需要重建索引或者进行替换，我们可以使用相同的 index API 进行实现，在 [索引文档](#) 中已经进行了讨论。

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

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

```
{
  "_index" : "website",
  "_type" : "blog",
  "_id" : "123",
  "_version" : 2,
  "created": false ( a)
}
```

(a) created 标志设置成 false，是因为相同的索引、类型和 ID 的文档已经存在。

在内部，Elasticsearch 已将旧文档标记为已删除，并增加一个全新的文档。尽管你不能再对旧版本的文档进行访问，但它并不会立即消失。当继续索引更多的数据，Elasticsearch 会在后台清理这些已删除文档。

在本章的后面部分，我们会介绍 update API，这个 API 可以用于 partial updates to a document。虽然它似乎对文档直接进行了修改，但实际上 Elasticsearch 按前述完全相同方式执行以下过程：

- 从旧文档构建 JSON
- 更改该 JSON
- 删除旧文档
- 索引一个新文档

唯一的区别在于，update API 仅仅通过一个客户端请求来实现这些步骤，而不需要单独的 get 和 index 请求。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-08

- 1. 创建新文档

1. 创建新文档

当我们索引一个文档，怎么确认我们正在创建一个完全新的文档，而不是覆盖现有的呢？

请记住，`_index`、`_type` 和 `_id` 的组合可以唯一标识一个文档。所以，确保创建一个新文档的最简单办法是，使用索引请求的 `POST` 形式让 Elasticsearch 自动生成唯一 `_id`：

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

然而，如果已经有自己的 `_id`，那么我们必须告诉 Elasticsearch，只有在相同的 `_index`、`_type` 和 `_id` 不存在时才接受我们的索引请求。这里有两种方式，他们做的实际是相同的事情。使用哪种，取决于哪种使用起来更方便。

第一种方法使用 `op_type` 查询-字符串参数：

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

第二种方法是在 URL 末端使用 `_create`：

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

如果创建新文档的请求成功执行，Elasticsearch 会返回元数据和一个 201 Created 的 HTTP 响应码。

另一方面，如果具有相同的 `_index`、`_type` 和 `_id` 的文档已经存在，Elasticsearch 将会返回 `409 Conflict` 响应码，以及如下的错误信息：

```
{  
  "error": {  
    "root_cause": [  
      {  
        "type": "document_already_exists_exception",  
        "reason": "[blog][123]: document already exists",  
        "shard": "0",  
        "index": "website"  
      }  
    ],  
    "type": "document_already_exists_exception",  
    "reason": "[blog][123]: document already exists",  
    "shard": "0",  
    "index": "website"  
  },  
  "status": 409  
}
```

- 1. 删除文档

1. 删除文档

删除文档的语法和我们所知道的规则相同，只是使用 DELETE 方法

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

如果找到该文档，Elasticsearch 将要返回一个 200 ok 的 HTTP 响应码，和一个类似以下结构的响应体。注意，字段 _version 值已经增加：

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

如果文档没有找到，我们将得到 404 Not Found 的响应码和类似这样的响应体：
即使文档不存在（Found 是 false），_version 值仍然会增加。这是 Elasticsearch 内部记录本的一部分，用来确保这些改变在跨多节点时以正确的顺序执行。

Note

正如已经在[更新整个文档](#)中提到的，删除文档不会立即将文档从磁盘中删除，只是将文档标记为已删除状态。随着你不断的索引更多的数据，Elasticsearch 将会在后台清理标记为已删除的文档。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-08

- 1. 处理冲突

1. 处理冲突

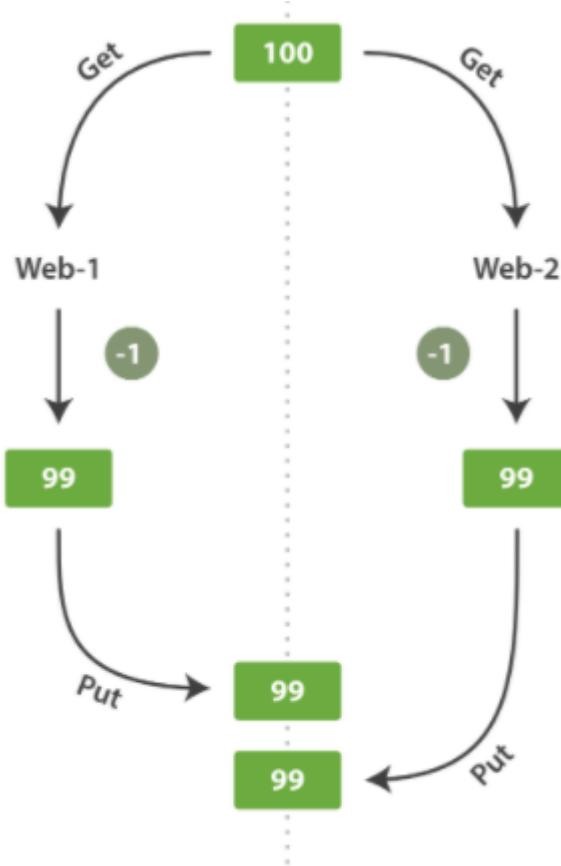
当我们使用 index API 更新文档，可以一次性读取原始文档，做我们的修改，然后重新索引整个文档。最近的索引请求将获胜：无论最后哪一个文档被索引，都将被唯一存储在 Elasticsearch 中。如果其他人同时更改这个文档，他们的更改将丢失。

很多时候这是没有问题的。也许我们的主数据存储是一个关系型数据库，我们只是将数据复制到 Elasticsearch 中并使其可被搜索。也许两个人同时更改相同的文档的几率很小。或者对于我们的业务来说偶尔丢失更改并不是很严重的问题。

但有时丢失了一个变更就是非常严重的。试想我们使用 Elasticsearch 存储我们网上商城商品库存的数量，每次我们卖一个商品的时候，我们在 Elasticsearch 中将库存数量减少。

有一天，管理层决定做一次促销。突然地，我们一秒要卖好几个商品。假设有两个 web 程序并行运行，每一个都同时处理所有商品的销售，如图 Figure 7，“Consequence of no concurrency control” 所示。

Figure 7. Consequence of no concurrency control



web_1 对 stock_count 所做的更改已经丢失，因为 web_2 不知道它的 stock_count 的拷贝已经过期。结果我们会认为有超过商品的实际数量的库存，因为卖给顾客的库存商品并不存在，我们将让他们非常失望。

变更越频繁，读数据和更新数据的间隙越长，也就越可能丢失变更。

在数据库领域中，有两种方法通常被用来确保并发更新时变更不会丢失：

悲观并发控制

这种方法被关系型数据库广泛使用，它假定有变更冲突可能发生，因此阻塞访问资源以防止冲突。一个典型的例子是读取一行数据之前先将其锁住，确保只有放置锁的线程能够对这行数据进行修改。

乐观并发控制

Elasticsearch 中使用的这种方法假定冲突是不可能发生的，并且不会阻塞正在尝试的操作。然而，如果源数据在读写当中被修改，更新将会失败。应用程序接下来将决定该如何解决冲突。例如，可以重试更新、使用新的数据、或者将相关情况报告给用户。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-08

- 1. 乐观并发控制
- 2. 通过外部系统使用版本控制

1. 乐观并发控制

Elasticsearch 是分布式的。当文档创建、更新或删除时，新版本的文档必须复制到集群中的其他节点。Elasticsearch 也是异步和并发的，这意味着这些复制请求被并行发送，并且到达目的地时也许顺序是乱的。Elasticsearch 需要一种方法确保文档的旧版本不会覆盖新的版本。

当我们之前讨论 `index`，`GET` 和 `delete` 请求时，我们指出每个文档都有一个 `_version`（版本）号，当文档被修改时版本号递增。Elasticsearch 使用这个 `_version` 号来确保变更以正确顺序得到执行。如果旧版本的文档在新版本之后到达，它可以被简单的忽略。

我们可以利用 `_version` 号来确保应用中相互冲突的变更不会导致数据丢失。我们通过指定想要修改文档的 `version` 号来达到这个目的。如果该版本不是当前版本号，我们的请求将会失败。

让我们创建一个新的博客文章：

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

响应体告诉我们，这个新创建的文档 `_version` 版本号是 1。现在假设我们想编辑这个文档：我们加载其数据到 web 表单中，做一些修改，然后保存新的版本。

首先我们检索文档：

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

响应体包含相同的 `_version` 版本号 1：

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

现在，当我们尝试通过重建文档的索引来保存修改，我们指定 `version` 为我们的修改会被应用的版本：

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

(a) 我们想这个在我们索引中的文档只有现在的 `_version` 为 1 时，本次更新才能成功。

此请求成功，并且响应体告诉我们 `_version` 已经递增到 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，仍然指定 `version=1`，Elasticsearch 返回 409 Conflict HTTP 响应码，和一个如下所示的响应体：

```
{
  "error": {
    "root_cause": [
      {
        "type": "version_conflict_engine_exception",
        "reason": "[blog][1]: version conflict, current [2], provided [1]"
        "index": "website",
        "shard": "3"
      }
    ],
    "type": "version_conflict_engine_exception",
    "reason": "[blog][1]: version conflict, current [2], provided [1]",
    "index": "website",
    "shard": "3"
  },
  "status": 409
}
```

这告诉我们在 Elasticsearch 中这个文档的当前 `_version` 号是 2，但我们指定的更新版本号为 1。

我们现在怎么做取决于我们的应用需求。我们可以告诉用户说其他人已经修改了文档，并且在再次保存之前检查这些修改内容。或者，在之前的商品 `stock_count` 场景，我们可以获取到最新的文档并尝试重新应用这些修改。

所有文档的更新或删除 API，都可以接受 `version` 参数，这允许你在代码中使用乐观的并发控制，这是一种明智的做法。

2. 通过外部系统使用版本控制

一个常见的设置是使用其它数据库作为主要的数据存储，使用 Elasticsearch 做数据检索，这意味着主数据库的所有更改发生时都需要被复制到 Elasticsearch，如果多个进程负责这一数据同步，你可能遇到类似于之前描述的并发问题。

如果你的主数据库已经有了版本号 — 或一个能作为版本号的字段值比如 `timestamp` — 那么你就可以在 Elasticsearch 中通过增加 `version_type=external` 到查询字符串的方式重用这些相同的版本号， 版本号必须是大于零的整数， 且小于 `9.2E+18` — 一个 Java 中 `long` 类型的正值。

外部版本号的处理方式和我们之前讨论的内部版本号的处理方式有些不同， Elasticsearch 不是检查当前 `_version` 和请求中指定的版本号是否相同， 而是检查当前 `_version` 是否 小于 指定的版本号。如果请求成功， 外部的版本号作为文档的新 `_version` 进行存储。

外部版本号不仅在索引和删除请求是可以指定， 而且在 创建 新文档时也可以指定。

例如， 要创建一个新的具有外部版本号 5 的博客文章， 我们可以按以下方法进行：

```
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` 版本号是 5 :

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

现在我们更新这个文档， 指定一个新的 version 号是 10 :

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

请求成功并将当前 `_version` 设为 10 :

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

如果你要重新运行此请求时， 它将会失败，并返回像我们之前看到的同样的冲突错误， 因为指定的外部版本号不大于 Elasticsearch 的当前版本号。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-08

- 1. 文档的部分更新
- 2. 使用脚本部分更新文档
- 3. 更新的文档可能尚不存在
- 4. 更新和冲突

1. 文档的部分更新

在更新整个文档，我们已经介绍过更新一个文档的方法是检索并修改它，然后重新索引整个文档，这的确如此。然而，使用 `update API` 我们还可以部分更新文档，例如在某个请求时对计数器进行累加。

我们也介绍过文档是不可变的：他们不能被修改，只能被替换。`update API` 必须遵循同样的规则。从外部来看，我们在一个文档的某个位置进行部分更新。然而在内部，`update API` 简单使用与之前描述相同的 检索-修改-重建索引 的处理过程。区别在于这个过程发生在分片内部，这样就避免了多次请求的网络开销。通过减少检索和重建索引步骤之间的时间，我们也减少了其他进程的变更带来冲突的可能性。

`update` 请求最简单的一种形式是接收文档的一部分作为 `doc` 的参数，它只是与现有的文档进行合并。对象被合并到一起，覆盖现有的字段，增加新的字段。例如，我们增加字段 `tags` 和 `views` 到我们的博客文章，如下所示：

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

如果请求成功，我们看到类似于 `index` 请求的响应：

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

检索文档显示了更新后的 `_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" ], ( a )
    "views": 0 ( a )
  }
}
```

(a) 新的字段已被添加到 `_source` 中

2. 使用脚本部分更新文档

脚本可以在 update API中用来改变 `_source` 的字段内容， 它在更新脚本中称为 `ctx._source` 。 例如，我们可以使用脚本来增加博客文章中 `views` 的数量：

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

用 Groovy 脚本编程

对于那些 API 不能满足需求的情况，Elasticsearch 允许你使用脚本编写自定义的逻辑。许多API都支持脚本的使用，包括搜索、排序、聚合和文档更新。脚本可以作为请求的一部分被传递，从特殊的 `.scripts` 索引中检索，或者从磁盘加载脚本。

默认的脚本语言是 Groovy，一种快速表达的脚本语言，在语法上与 JavaScript 类似。它在 Elasticsearch V1.3.0 版本首次引入并运行在沙盒中，然而 Groovy 脚本引擎存在漏洞，允许攻击者通过构建 Groovy 脚本，在 Elasticsearch Java VM 运行时脱离沙盒并执行 shell 命令。

因此，在版本 v1.3.8 、 1.4.3 和 V1.5.0 及更高的版本中，它已经被默认禁用。此外，您可以通过设置集群中的所有节点的 config/elasticsearch.yml 文件来禁用动态 Groovy 脚本：

```
script.groovy.sandbox.enabled: false
```

这将关闭 Groovy 沙盒，从而防止动态 Groovy 脚本作为请求的一部分被接受，或者从特殊的 `.scripts` 索引中被检索。当然，你仍然可以使用存储在每个节点的 config/scripts/ 目录下的 Groovy 脚本。

如果你的架构和安全性不需要担心漏洞攻击，例如你的 Elasticsearch 终端仅暴露和提供给可信赖的应用，当它是你的应用需要的特性时，你可以选择重新启用动态脚本。

你可以在 scripting reference documentation 获取更多关于脚本的资料。

我们也可以通过使用脚本给 `tags` 数组添加一个新的标签。在这个例子中，我们指定新的标签作为参数，而不是硬编码到脚本内部。这使得 Elasticsearch 可以重用这个脚本，而不是每次我们想添加标签时都要对新脚本重新编译：

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

获取文档并显示最后两次请求的效果：

```
{
  "_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"], (a)
    "views": 1 (b)
  }
}
```

(a) search 标签已追加到 tags 数组中。
(b) views 字段已递增。

我们甚至可以选择通过设置 ctx.op 为 delete 来删除基于其内容的文档：

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

3. 更新的文档可能尚不存在

假设我们需要在 Elasticsearch 中存储一个页面访问量计数器。每当有用户浏览网页，我们对该页面的计数器进行累加。但是，如果它是一个新网页，我们不能确定计数器已经存在。如果我们尝试更新一个不存在的文档，那么更新操作将会失败。

在这样的情况下，我们可以使用 upsert 参数，指定如果文档不存在就应该先创建它：

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

我们第一次运行这个请求时，`upsert` 值作为新文档被索引，初始化 `views` 字段为 1。在后续的运行中，由于文档已经存在，`script` 更新操作将替代 `upsert` 进行应用，对 `views` 计数器进行累加。

4. 更新和冲突

在本节的介绍中，我们说明 检索 和 重建索引 步骤的间隔越小，变更冲突的机会越小。但是它并不能完全消除冲突的可能性。还是有可能在 `update` 设法重新索引之前，来自另一进程的请求修改了文档。

为了避免数据丢失，`update API` 在 检索 步骤时检索得到文档当前的 `_version` 号，并传递版本号到 重建索引 步骤的 `index` 请求。如果另一个进程修改了处于检索和重新索引步骤之间的文档，那么 `_version` 号将不匹配，更新请求将会失败。

对于部分更新的很多使用场景，文档已经被改变也没有关系。例如，如果两个进程都对页面访问量计数器进行递增操作，它们发生的先后顺序其实不太重要；如果冲突发生了，我们唯一需要做的就是尝试再次更新。

这可以通过设置参数 `retry_on_conflict` 来自动完成，这个参数规定了失败之前 `update` 应该重试的次数，它的默认值为 0。

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

(a) 失败之前重试该更新5次。

在增量操作无关顺序的场景，例如递增计数器等这个方法十分有效，但是在其他情况下变更的顺序是非常重要的。类似 [index API](#)，`update API` 默认采用最终写入生效的方案，但它也接受一个 `version` 参数来允许你使用 [optimistic concurrency control](#) 指定想要更新文档的版本。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-09

- 1. 取回多个文档

1. 取回多个文档

Elasticsearch 的速度已经很快了，但甚至能更快。将多个请求合并成一个，避免单独处理每个请求花费的网络延时和开销。如果你需要从 Elasticsearch 检索很多文档，那么使用 `multi-get` 或者 `mget API` 来将这些检索请求放在一个请求中，将比逐个文档请求更快地检索到全部文档。

`mget API` 要求有一个 `docs` 数组作为参数，每个元素包含需要检索文档的元数据，包括 `_index`、`_type` 和 `_id`。如果你想检索一个或者多个特定的字段，那么你可以通过 `_source` 参数来指定这些字段的名字：

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

该响应体也包含一个 `docs` 数组，对于每一个在请求中指定的文档，这个数组中都包含有一个对应的响应，且顺序与请求中的顺序相同。其中的每一个响应都和使用单个 `get request` 请求所得到的响应体相同：

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

如果想检索的数据都在相同的 `_index` 中（甚至相同的 `_type` 中），则可以在 URL 中指定默认的 `/_index` 或者默认的 `/_index/_type`。

你仍然可以通过单独请求覆盖这些值：

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

事实上，如果所有文档的 `_index` 和 `_type` 都是相同的，你可以只传一个 `ids` 数组，而不是整个 `docs` 数组：

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

注意，我们请求的第二个文档是不存在的。我们指定类型为 `blog`，但是文档 ID 1 的类型是 `pageviews`，这个不存在的情况将在响应体中被报告：

```
{  
  "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  
    }  
  ]  
}
```

(a) 未找到该文档

事实上第二个文档未能找到并不妨碍第一个文档被检索到。每个文档都是单独检索和报告的。

Note

即使有某个文档没有找到，上述请求的 HTTP 状态码仍然是 200。事实上，即使请求没有找到任何文档，它的状态码依然是 200 -- 因为 mget 请求本身已经成功执行。为了确定某个文档查找是成功或者失败，你需要检查 found 标记。

Copyright © Songbai Yang all right reserved, powered by Gitbook 该文件修订时间：2021-03-09

- 1. 代价较小的批量操作
 - 1.1. 不要重复指定Index和Type
 - 1.2. 多大是太大?

1. 代价较小的批量操作

与 `mget` 可以使我们一次取回多个文档同样的方式，`bulk` API 允许在单个步骤中进行多次 `create`、`index`、`update` 或 `delete` 请求。如果你需要索引一个数据流比如日志事件，它可以排队和索引数百或数千批次。

`bulk` 与其他的请求体格式稍有不同，如下所示：

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

这种格式类似一个有效的单行 JSON 文档流，它通过换行符(`\n`)连接到一起。注意两个要点：

- 每行一定要以换行符(`\n`)结尾，包括最后一行。这些换行符被用作一个标记，可以有效分隔行。
- 这些行不能包含未转义的换行符，因为他们将会对解析造成干扰。这意味着这个 JSON 不能使用 `pretty` 参数打印。

Tip

在 [为什么是有趣的格式？](#) 中，我们解释为什么 bulk API 使用这种格式。

`action/metadata` 行指定 哪一个文档 做 什么操作 。

`action` 必须是以下选项之一：

`create`

如果文档不存在，那么就创建它。详情请见 [创建新文档](#)。

`index`

创建一个新文档或者替换一个现有的文档。详情请见 [索引文档](#) 和 [更新整个文档](#)。

`update`

部分更新一个文档。详情请见 [文档的部分更新](#)。

`delete`

删除一个文档。详情请见 [删除文档](#)。

`metadata`

应该指定被索引、创建、更新或者删除的文档的 `_index`、`_type` 和 `_id` 。

例如，一个 `delete` 请求看起来是这样的：

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

`request body` 行由文档的 `_source` 本身组成—文档包含的字段和值。它是 `index` 和 `create` 操作所必需的，这是有道理的：你必须提供文档以索引。它也是 `update` 操作所必需的，并且应该包含你传递给 `update API` 的相同请求体：`doc`、`upsert`、`script` 等等。删除操作不需要 `request body` 行。

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

如果不指定 _id , 将会自动生成一个 ID :

```
{ "index": { "_index": "website", "_type": "blog" }}  
{ "title": "My second blog post" }
```

为了把所有的操作组合在一起, 一个完整的 bulk 请求 有以下形式:

```
POST /_bulk  
{ "delete": { "_index": "website", "_type": "blog", "_id": "123" }} (a)  
{ "create": { "_index": "website", "_type": "blog", "_id": "123" }}  
{ "title": "My first blog post" }  
{ "index": { "_index": "website", "_type": "blog" }}  
{ "title": "My second blog post" }  
{ "update": { "_index": "website", "_type": "blog", "_id": "123", "_retry_on_conflict": 3 } }  
{ "doc" : {"title" : "My updated blog post"} } (b)
```

(a) 请注意 delete 动作不能有请求体, 它后面跟着的是另外一个操作。

(b) 谨记最后一个换行符不要落下。

这个 Elasticsearch 响应包含 items 数组, 这个数组的内容是以请求的顺序列出来的每个请求的结果。

```
{
  "took": 4,
  "errors": false, ( a )
  "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
      }}
  ]
}
```

(a) 所有的子请求都成功完成。

每个子请求都是独立执行，因此某个子请求的失败不会对其他子请求的成功与否造成影响。如果其中任何子请求失败，最顶层的 error 标志被设置为 true，并且在相应的请求报告出错误明细：

```
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" }
```

在响应中，我们看到 create 文档 123 失败，因为它已经存在。但是随后的 index 请求，也是对文档 123 操作，就成功了：

```
{  
    "took": 3,  
    "errors": true, ( a)  
    "items": [  
        { "create": {  
            "_index": "website",  
            "_type": "blog",  
            "_id": "123",  
            "status": 409, ( b)  
            "error": "DocumentAlreadyExistsException ( c)  
[[website][4] [blog][123]:  
document already exists]"  
        }},  
        { "index": {  
            "_index": "website",  
            "_type": "blog",  
            "_id": "123",  
            "_version": 5,  
            "status": 200 ( d)  
        }}  
    ]  
}
```

- (a) 一个或者多个请求失败。
- (b) 这个请求的HTTP状态码报告为 409 CONFLICT 。
- (c) 解释为什么请求失败的错误信息。
- (d) 第二个请求成功，返回 HTTP 状态码 200 OK 。

这也意味着 bulk 请求不是原子的：不能用它来实现事务控制。每个请求是单独处理的，因此一个请求的成功或失败不会影响其他的请求。

1.1. 不要重复指定Index和Type

也许你正在批量索引日志数据到相同的 index 和 type 中。但为每一个文档指定相同的元数据是一种浪费。相反，可以像 mget API 一样，在 bulk 请求的 URL 中接收默认的 _index 或者 _index/_type

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

你仍然可以覆盖元数据行中的 _index 和 _type，但是它将使用 URL 中的这些元数据值作为默认值：

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

1.2. 多大是太大？

整个批量请求都需要由接收到请求的节点加载到内存中，因此该请求越大，其他请求所能获得的内存就越少。批量请求的大小有一个最佳值，大于这个值，性能将不再提升，甚至会下降。但是最佳值不是一个固定的值。它完全取决于硬件、文

档的大小和复杂度、索引和搜索的负载的整体情况。

幸运的是，很容易找到这个 最佳点：通过批量索引典型文档，并不断增加批量大小进行尝试。当性能开始下降，那么你的批量大小就太大了。一个好的办法是开始时将 1,000 到 5,000 个文档作为一个批次，如果你的文档非常大，那么就减少批量的文档个数。

密切关注你的批量请求的物理大小往往非常有用，一千个 1KB 的文档是完全不同于一千个 1MB 文档所占的物理大小。一个好的批量大小在开始处理后所占用的物理大小约为 5-15 MB。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-09

- 1. 分布式文档存储

1. 分布式文档存储

在前面的章节，我们介绍了如何索引和查询数据，不过我们忽略了很多底层的技术细节，例如文件是如何分布到集群的，又是如何从集群中获取的。Elasticsearch 本意就是隐藏这些底层细节，让我们好专注在业务开发中，所以其实你不必了解这么深入也无妨。

在这个章节中，我们将深入探索这些核心的技术细节，这能帮助你更好地理解数据如何被存储到这个分布式系统中。

NOTE

这个章节包含了一些高级话题，上面也提到过，就算你不记住和理解所有的细节仍然能正常使用 Elasticsearch。如果你有兴趣的话，这个章节可以作为你的课外兴趣读物，扩展你的知识面。

如果你在阅读这个章节的时候感到很吃力，也不用担心。这个章节仅仅只是用来告诉你 Elasticsearch 是如何工作的，将来在工作中如果你需要用到这个章节提供的知识，可以再回过头来翻阅。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-09

- 1. 路由一个文档到一个分片中

1. 路由一个文档到一个分片中

当索引一个文档的时候，文档会被存储到一个主分片中。Elasticsearch 如何知道一个文档应该存放到哪个分片中呢？当我们创建文档时，它如何决定这个文档应当被存储在分片 1 还是分片 2 中呢？

首先这肯定不会是随机的，否则将来要获取文档的时候我们就不知道从何处寻找了。实际上，这个过程是根据下面这个公式决定的：

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

`routing` 是一个可变值，默认是文档的 `_id`，也可以设置成一个自定义的值。`routing` 通过 `hash` 函数生成一个数字，然后这个数字再除以 `number_of_primary_shards`（主分片的数量）后得到余数。这个分布在 0 到 `number_of_primary_shards-1` 之间的余数，就是我们所寻求的文档所在分片的位置。

这就解释了为什么我们要在创建索引的时候就确定好主分片的数量 并且永远不会改变这个数量：因为如果数量变化了，那么所有之前路由的值都会无效，文档再也找不到了。

NOTE

你可能觉得由于 Elasticsearch 主分片数量是固定的会使索引难以进行扩容。实际上当你需要时有很多技巧可以轻松实现扩容。我们将会在扩容设计一章中提到更多有关水平扩展的内容。

所有的文档 API（`get`、`index`、`delete`、`bulk`、`update` 以及 `mget`）都接受一个叫做 `routing` 的路由参数，通过这个参数我们可以自定义文档到分片的映射。一个自定义的路由参数可以用来确保所有相关的文档——例如所有属于同一个用户的文档——都被存储到同一个分片中。我们也会在扩容设计这一章中详细讨论为什么会有这样一种需求。

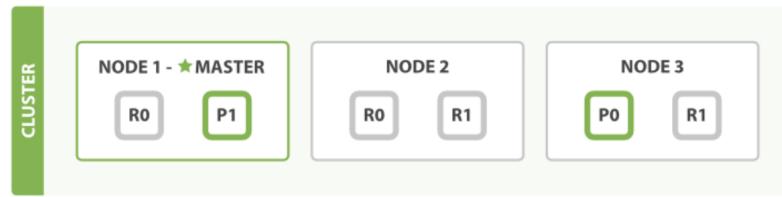
Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-09

- 1. 主分片和副本分片如何交互

1. 主分片和副本分片如何交互

为了说明目的, 我们假设有一个集群由三个节点组成。它包含一个叫 blogs 的索引, 有两个主分片, 每个主分片有两个副本分片。相同分片的副本不会放在同一节点, 所以我们的集群看起来像 Figure 8, “有三个节点和一个索引的集群”。

Figure 8. 有三个节点和一个索引的集群



我们可以发送请求到集群中的任一节点。每个节点都有能力处理任意请求。每个节点都知道集群中任一文档位置, 所以可以直接将请求转发到需要的节点上。在下面的例子中, 将所有的请求发送到 Node 1, 我们将其称为 协调节点 (coordinating node)。

Note

当发送请求的时候, 为了扩展负载, 更好的做法是轮询集群中所有的节点。

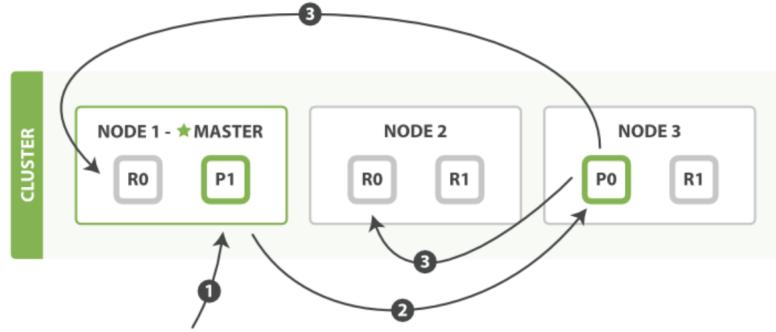
Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间: 2021-03-09

- 1. 新建、索引和删除文档

1. 新建、索引和删除文档

新建、索引和删除 请求都是 写 操作， 必须在主分片上面完成之后才能被复制到相关的副本分片， 如下图所示 Figure 9, “新建、索引和删除单个文档”.

Figure 9. 新建、索引和删除单个文档



以下是在主副分片和任何副本分片上面 成功新建，索引和删除文档所需要的步骤顺序：

客户端向 Node 1 发送新建、索引或者删除请求。 1 节点使用文档的 `_id` 确定文档属于分片 0 。请求会被转发到 Node 3，因为分片 0 的主分片 目前被分配在 Node 3 上。 2 Node 3 在主分片上面执行请求。如果成功了，它将请求并行转发到 Node 1 和 Node 2 的副本分片上。 3 一旦所有的副本分片都报告成功, Node 3 将向协调节点报告成功，协调节点向客户端报告成功。在客户端收到成功响应时，文档变更已经在主分片和所有副本分片执行完成，变更是安全的。

有一些可选的请求参数允许您影响这个过程，可能以数据安全为代价提升性能。这些选项很少使用，因为Elasticsearch已经很快，但是为了完整起见，在这里阐述如下：

consistency

`consistency`, 即一致性。在默认设置下，即使仅仅是在试图执行一个写操作之前，主分片都会要求 必须要有 规定数量(`quorum`) (或者换种说法，也即必须要有 大多数) 的分片副本处于活跃可用状态， 才会去执行写操作(其中分片副本可以是 主分片或者副本分片)。这是为了避免在发生网络分区故障 (network partition) 的时候进行写操作，进而导致数据不一致。规定数量即：

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

`consistency` 参数的值可以设为 `one` (只要主分片状态 `ok` 就允许执行写操作) , `all` (必须要主分片和所有副本分片的状态没问题才允许执行写操作) , 或 `quorum` 。默认值为 `quorum` , 即大多数的分片副本状态没问题就允许执行写操作。

注意，规定数量 的计算公式中 `number_of_replicas` 指的是在索引设置中的设定副本分片数， 而不是指当前处理活动状态的副本分片数。如果你的索引设置中指定了当前索引拥有三个副本分片，那规定数量的计算结果即：

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

如果此时你只启动两个节点，那么处于活跃状态的分片副本数量就达不到规定数量，也因此您将无法索引和删除任何文档。

timeout

如果没有足够的副本分片会发生什么？ Elasticsearch 会等待，希望更多的分片出现。默认情况下，它最多等待1分钟。如果你需要，你可以使用 `timeout` 参数使它更早终止：`100ms` 是100毫秒，`30s` 是30秒。

NOTE

新索引默认有 1 个副本分片，这意味着为满足 规定数量 应该 需要两个活动的分片副本。但是，这些默认的设置会阻止我们在单一节点上做任何事情。为了避免这个问题，要求只有当 `number_of_replicas` 大于1的时候，规定数量才会执行。

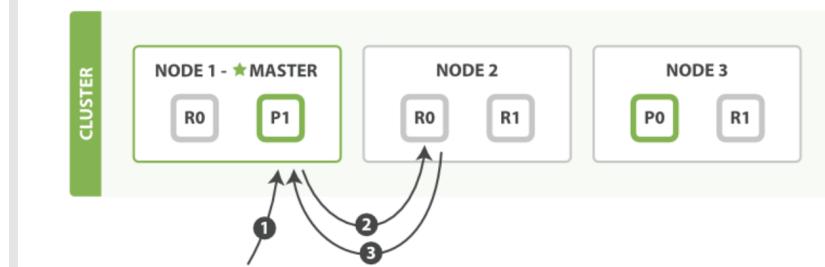
Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-10

- 1. 取回一个文档

1. 取回一个文档

可以从主分片或者从其它任意副本分片检索文档，如下图所示 Figure 10, “取回单个文档”。

Figure 10. 取回单个文档



以下是从主分片或者副本分片检索文档的步骤顺序：

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

在处理读取请求时，协调结点在每次请求的时候都会通过轮询所有的副本分片来达到负载均衡。

在文档被检索时，已经被索引的文档可能已经存在于主分片上但是还没有复制到副本分片。在这种情况下，副本分片可能会报告文档不存在，但是主分片可能成功返回文档。一旦索引请求成功返回给用户，文档在主分片和副本分片都是可用的。

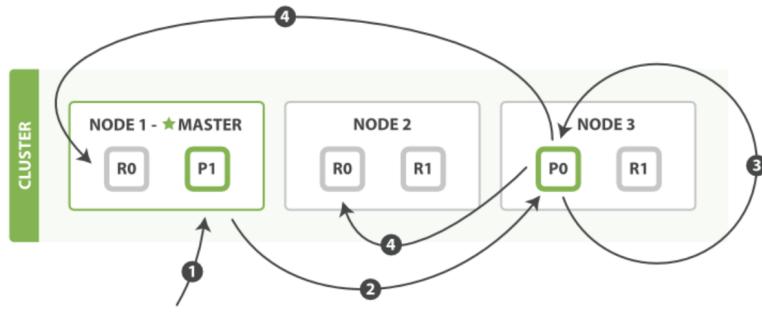
Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-10

- 1. 局部更新文档

1. 局部更新文档

如 Figure 11, “局部更新文档” 所示, update API 结合了先前说明的读取和写入模式。

Figure 11. 局部更新文档



以下是部分更新一个文档的步骤:

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

update API 还接受在 新建、索引和删除文档 章节中介绍的 `routing`、
`replication`、`consistency` 和 `timeout` 参数。

基于文档的复制

当主分片把更改转发到副本分片时, 它不会转发更新请求。相反, 它转发完整文档的新版本。请记住, 这些更改将会异步转发到副本分片, 并且不能保证它们以发送它们相同的顺序到达。如果 Elasticsearch 仅转发更改请求, 则可能以错误的顺序应用更改, 导致得到损坏的文档。

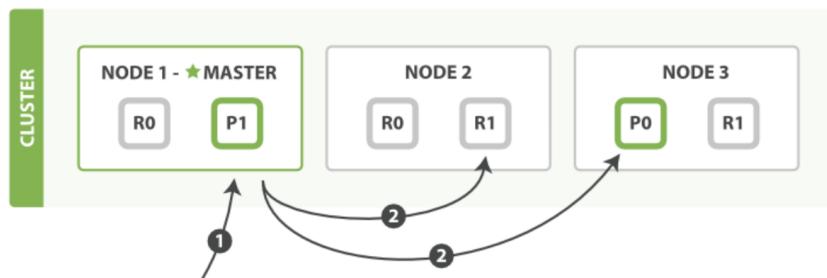
Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间: 2021-03-10

- 1. 多文档模式
 - 1.1. 为什么是有趣的格式

1. 多文档模式

`mget` 和 `bulk` API 的模式类似于单文档模式。区别在于协调节点知道每个文档存在于哪个分片中。它将整个多文档请求分解成每个分片的多文档请求，并且将这些请求并行转发到每个参与节点。

协调节点一旦收到来自每个节点的应答，就将每个节点的响应收集整理成单个响应，返回给客户端，如 Figure 12，“使用 `mget` 取回多个文档”所示。



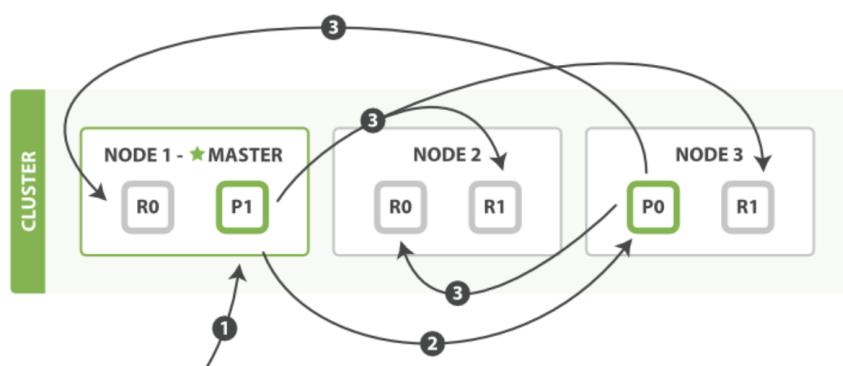
以下是使用单个 `mget` 请求取回多个文档所需的步骤顺序：

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

可以对 `docs` 数组中每个文档设置 `routing` 参数。

`bulk` API，如 Figure 13，“使用 `bulk` 修改多个文档”所示，允许在单个批量请求中执行多个创建、索引、删除和更新请求。

Figure 13. 使用 `bulk` 修改多个文档



`bulk` API 按如下步骤顺序执行：

- 1 客户端向 Node 1 发送 `bulk` 请求。
- 2 Node 1 为每个节点创建一个批量请求，并将这些请求并行转发到每个包含主分片的节点主机。
- 3 主分片一个接一个按顺序执行每个操作。当每个操作成功时，主分片并行

转发新文档（或删除）到副本分片，然后执行下一个操作。一旦所有的副本分片报告所有操作成功，该节点将向协调节点报告成功，协调节点将这些响应收集整理并返回给客户端。

bulk API 还可以在整个批量请求的最顶层使用 `consistency` 参数，以及在每个请求中的元数据中使用 `routing` 参数。

1.1. 为什么是有趣的格式

当我们早些时候在[代价较小的批量操作](#)章节了解批量请求时，您可能会问自己，“为什么 bulk API 需要有换行符的有趣格式，而不是发送包装在 JSON 数组中的请求，例如 mget API？”。

为了回答这一点，我们需要解释一点背景：在批量请求中引用的每个文档可能属于不同的主分片，每个文档可能被分配给集群中的任何节点。这意味着批量请求 bulk 中的每个操作都需要被转发到正确节点上的正确分片。

如果单个请求被包装在 JSON 数组中，那就意味着我们需要执行以下操作：

- 将 JSON 解析为数组（包括文档数据，可以非常大）
- 查看每个请求以确定应该去哪个分片
- 为每个分片创建一个请求数组
- 将这些数组序列化为内部传输格式
- 将请求发送到每个分片

这是可行的，但需要大量的 RAM 来存储原本相同的数据的副本，并将创建更多的数据结构，Java 虚拟机（JVM）将不得不花费时间进行垃圾回收。

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

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

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-10

- 1. 搜索——最基本的工具

1. 搜索——最基本的工具

现在，我们已经学会了如何使用 Elasticsearch 作为一个简单的 NoSQL 风格的分布式文档存储系统。我们可以将一个 JSON 文档扔到 Elasticsearch 里，然后根据 ID 检索。但 Elasticsearch 真正强大之处在于可以从无规律的数据中找出有意义的信息——从“大数据”到“大信息”。

Elasticsearch 不只会存储 (*stores*) 文档，为了能被搜索到也会为文档添加索引 (*indexes*)，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。

文档中的每个字段都将被索引并且可以被查询。不仅如此，在简单查询时，Elasticsearch 可以使用所有 (all) 这些索引字段，以惊人的速度返回结果。这是你永远不会考虑用传统数据库去做的一些事情。

搜索 (search) 可以做到：

- 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，找出所有匹配关键字的文档并按照相关性 (*relevance*) 排序后返回结果。
- 以上二者兼而有之。

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：

映射 (Mapping)

描述数据在每个字段内如何存储

分析 (Analysis)

全文是如何处理使之可以被搜索的

领域特定查询语言 (Query DSL)

Elasticsearch 中强大灵活的查询语言

以上提到的每个点都是一个大话题，我们将在深入搜索一章详细阐述它们。

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

本章节的测试数据可以在这里找到：

<https://gist.github.com/clintongormley/8579281>。

你可以把这些命令复制到终端中执行来实践本章的例子。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-10

- [1. 空搜索](#)
 - [1.1. hits](#)
 - [1.2. took](#)
 - [1.3. shards](#)
 - [1.4. timeout](#)

1. 空搜索

搜索API的最基础的形式是没有指定任何查询的空搜索，它简单地返回集群中所有索引下的所有文档：

```
GET /_search
```

返回的结果（为了界面简洁编辑过的）像这样：

```
{
  "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
}
```

1.1. hits

返回结果中最重要的部分是 `hits`，它包含 `total` 字段来表示匹配到的文档总数，并且一个 `hits` 数组包含所查询结果的前十个文档。

在 `hits` 数组中每个结果包含文档的 `_index`、`_type`、`_id`，加上 `_source` 字段。这意味着我们可以直接从返回的搜索结果中使用整个文档。这不像其他的搜索引擎，仅仅返回文档的 `ID`，需要你单独去获取文档。

每个结果还有一个 `_score`，它衡量了文档与查询的匹配程度。默认情况下，首先返回最相关的文档结果，就是说，返回的文档是按照 `_score` 降序排列的。在这个例子中，我们没有指定任何查询，故所有的文档具有相同的相关性，因此对所有的结果而言 `1` 是中性的 `_score`。

`max_score` 值是与查询所匹配文档的 `_score` 的最大值。

1.2. took

`took` 值告诉我们执行整个搜索请求耗费了多少毫秒。

1.3. shards

`_shards` 部分告诉我们在查询中参与分片的总数，以及这些分片成功了多少个失败了多少个。正常情况下我们不希望分片失败，但是分片失败是可能发生的。如果我们遭遇到一种灾难级别的故障，在这个故障中丢失了相同分片的原始数据和副本，那么对这个分片将没有可用副本对搜索请求作出响应。假若这样，Elasticsearch 将报告这个分片是失败的，但是会继续返回剩余分片的结果。

1.4. timeout

`timed_out` 值告诉我们查询是否超时。默认情况下，搜索请求不会超时。如果低响应时间比完成结果更重要，你可以指定 `timeout` 为 `10` 或者 `10ms`（10毫秒），或者 `1s`（1秒）：

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

在请求超时之前，Elasticsearch 将会返回已经成功从每个分片获取的结果。

警示!

应当注意的是 `timeout` 不是停止执行查询，它仅仅是告知正在协调的节点返回到目前为止收集的结果并且关闭连接。在后台，其他的分片可能仍在执行查询即使是结果已经被发送了。

使用超时是因为 SLA(服务等级协议)对你是很重要的，而不是因为想去中止长时间运行的查询。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-10

- 1. 多索引

1. 多索引

搜索API的最基础的形式是没有指定任何查询的空搜索，它简单地返回集群中所有索引下的所有文档：你有没有注意到之前的 `empty search` 的结果，不同类型的数据—`user` 和 `tweet` 来自不同的索引—`us` 和 `gb`？

搜索API的最基础的形式是没有指定任何查询的空搜索，它简单地返回集群中所有索引下的所有文档：如果不对某一特殊的索引或者类型做限制，就会搜索集群中的所有文档。Elasticsearch 转发搜索请求到每一个主分片或者副本分片，汇集查询出的前10个结果，并且返回给我们。

然而，经常的情况下，你想在一个或多个特殊的索引并且在一个或者多个特殊的类型中进行搜索。我们可以通过在URL中指定特殊的索引和类型达到这种效果，如下所示：

<code>/_search</code>	在所有的索引中搜索所有的类型
<code>/gb/_search</code>	在 <code>gb</code> 索引中搜索所有的类型
<code>/gb,us/_search</code>	在 <code>gb</code> 和 <code>us</code> 索引中搜索所有的文档
<code>/g*,u*/_search</code>	在任何以 <code>g</code> 或者 <code>u</code> 开头的索引中搜索所有的类型
<code>/gb/user/_search</code>	在 <code>gb</code> 索引中搜索 <code>user</code> 类型
<code>/gb,us/user,tweet/_search</code>	在 <code>gb</code> 和 <code>us</code> 索引中搜索 <code>user</code> 和 <code>tweet</code> 类型
<code>/_all/user,tweet/_search</code>	在所有的索引中搜索 <code>user</code> 和 <code>tweet</code> 类型

当在单一的索引下进行搜索的时候，Elasticsearch 转发请求到索引的每个分片中，可以是主分片也可以是副本分片，然后从每个分片中收集结果。多索引搜索恰好也是用相同的方式工作的一只是会涉及到更多的分片。

警示 !

搜索一个索引有五个主分片和搜索五个索引各有一个分片准确来说是等价的。Elasticsearch6版本之后

接下来，你将明白这种简单的方式如何灵活的根据需求的变化让扩容变得简单。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-03-10

- [1. 分页](#)

1. 分页

在之前的 空搜索 中说明了集群中有 14 个文档匹配了 (empty) query 。但是在 hits 数组中只有 10 个文档。如何才能看到其他的文档？

和 SQL 使用 LIMIT 关键字返回单个 page 结果的方法相同， Elasticsearch 接受 from 和 size 参数：

`size`
显示应该返回的结果数量， 默认是 10
`from` 显示应该跳过的初始结果数量， 默认是 0

如果每页展示 5 条结果，可以用下面方式请求得到 1 到 3 页的结果：

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

考虑到分页过深以及一次请求太多结果的情况，结果集在返回之前先进行排序。但请记住一个请求经常跨越多个分片，每个分片都产生自己的排序结果，这些结果需要进行集中排序以保证整体顺序是正确的。

在分布式系统中深度分页

理解为什么深度分页是有问题的，我们可以假设在一个有 5 个主分片的索引中搜索。当我们请求结果的第一页（结果从 1 到 10），每一个分片产生前 10 的结果，并且返回给协调节点，协调节点对 50 个结果排序得到全部结果的前 10 个。

现在假设我们请求第 1000 页—结果从 10001 到 10010。所有都以相同的方式工作除了每个分片不得不产生前 1000 个结果以外。然后协调节点对全部 5000 个结果排序最后丢弃掉这些结果中的 5000 个结果。

可以看到，在分布式系统中，对结果排序的成本随分页的深度成指数上升。这就是 web 搜索引擎对任何查询都不要返回超过 1000 个结果的原因。

Tip

在 [重新索引你的数据](#) 中解释了如何能够有效获取大量的文档。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-10

- [1. 轻量 搜索](#)
- [2. `_all`](#)
 - [2.1. 更复杂的查询](#)

1. 轻量 搜索

有两种形式的 搜索 API：一种是“轻量的”查询字符串 版本，要求在查询字符串中传递所有的参数，另一种是更完整的 请求体 版本，要求使用 JSON 格式和更丰富的查询表达式作为搜索语言。

查询字符串搜索非常适用于通过命令行做即席查询。例如，查询在 tweet 类型中 tweet 字段包含 elasticsearch 单词的所有文档：

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

下一个查询在 name 字段中包含 john 并且在 tweet 字段中包含 mary 的文档。
实际的查询就是这样

`+name:john +tweet:mary` 但是查询字符串参数所需要的 百分比编码（译者注：
URL编码）实际上更加难懂：

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

`+` 前缀表示必须与查询条件匹配。类似地，`-` 前缀表示一定不与查询条件
匹配。没有 `+` 或者 `-` 的所有其他条件都是可选的——匹配的越多，文档就越相
关。

2. `_all`

这个简单搜索返回包含 `mary` 的所有文档：

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

之前的例子中，我们在 `tweet` 和 `name` 字段中搜索内容。然而，这个查询的结果
在三个地方提到了 `mary`：

- 有一个用户叫做 Mary
- 6条微博发自 Mary
- 一条微博直接 @mary

Elasticsearch 是如何在三个不同的字段中查找到结果的呢？

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

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

这就好似增加了一个名叫 `_all` 的额外字段：

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

Tip

在刚开始开发一个应用时，`_all` 字段是一个很实用的特性。之后，你会发现如果搜索时用指定字段来代替 `_all` 字段，将会更好控制搜索结果。当 `_all` 字段不再有用的时候，可以将它置为失效，正如在元数据: `_all` 字段 中所解释的。

2.1. 更复杂的查询

下面的查询针对 `tweets` 类型，并使用以下的条件：

- `name` 字段中包含 `mary` 或者 `john`
- `date` 值大于 `2014-09-10`
- `_all` 字段包含 `aggregations` 或者 `geo`

查询字符串在做了适当的编码后，可读性很差：

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

从之前的例子中可以看出，这种 轻量 的查询字符串搜索效果还是挺让人惊喜的。它的查询语法在相关参考文档中有详细解释，以便简洁的表达很复杂的查询。对于通过命令做一次性查询，或者是在开发阶段，都非常方便。

但同时也可以看到，这种精简让调试更加晦涩和困难。而且很脆弱，一些查询字符串中很小的语法错误，像 `-`，`:`，`/` 或者 `"` 不匹配等，将会返回错误而不是搜索结果。

最后，查询字符串搜索允许任何用户在索引的任意字段上执行可能较慢且重量级的查询，这可能会暴露隐私信息，甚至将集群拖垮。

Tip

因为这些原因，不推荐直接向用户暴露查询字符串搜索功能，除非对于集群和数据来说非常信任他们。

相反，我们经常在生产环境中更多地使用功能全面的 `request body` 查询API，除了能完成以上所有功能，还有一些附加功能。但在到达那个阶段之前，首先需要了解数据在 Elasticsearch 中是如何被索引的。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-03-10

- 1. 映射和分析

1. 映射和分析

当摆弄索引里面的数据时，我们发现一些奇怪的事情。一些事情看起来被打乱了：在我们的索引中有12条推文，其中只有一条包含日期 `2014-09-15`，但是看一看下面查询命中的总数（`total`）：

```
GET /_search?q=2014          # 12 results
GET /_search?q=2014-09-15    # 12 results !
GET /_search?q=date:2014-09-15 # 1 result
GET /_search?q=date:2014      # 0 results !
```

为什么在 `_all` 字段查询日期返回所有推文，而在 `date` 字段只查询年份却没有返回结果？为什么我们在 `_all` 字段和 `date` 字段的查询结果有差别？

推测起来，这是因为数据在 `_all` 字段与 `date` 字段的索引方式不同。所以，通过请求 `gb` 索引中 `tweet` 类型的映射（或模式定义），让我们看一看 Elasticsearch 是如何解释我们文档结构的：

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

这将得到如下结果：

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

基于对字段类型的猜测，Elasticsearch 动态为我们产生了一个映射。这个响应告诉我们 `date` 字段被认为是 `date` 类型的。由于 `_all` 是默认字段，所以没有提及它。但是我们知道 `_all` 字段是 `string` 类型的。

所以 `date` 字段和 `string` 字段索引方式不同，因此搜索结果也不一样。这完全不令人吃惊。你可能会认为核心数据类型 `strings`、`numbers`、`Booleans` 和 `dates` 的索引方式有稍许不同。没错，他们确实稍有不同。

但是，到目前为止，最大的差异在于代表 精确值（它包括 string 字段）的字段和代表 全文 的字段。这个区别非常重要——它将搜索引擎和所有其他数据库区别开来。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-10

- 1. 精确值 VS 全文

1. 精确值 VS 全文

Elasticsearch 中的数据可以概括的分为两类：精确值和全文。

精确值 如它们听起来那样精确。例如日期或者用户 ID，但字符串也可以表示精确值，例如用户名或邮箱地址。对于精确值来讲， Foo 和 foo 是不同的， 2014 和 2014-09-15 也是不同的。

另一方面，全文 是指文本数据（通常以人类容易识别的语言书写），例如一个推文的内容或一封邮件的内容。

NOTE

全文通常是指非结构化的数据，但这里有一个误解：自然语言是高度结构化的。问题在于自然语言的规则是复杂的，导致计算机难以正确解析。例如，考虑这条语句：

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

它指的是月份还是人？

精确值很容易查询。结果是二进制的：要么匹配查询，要么不匹配。这种查询很容易用 SQL 表示：

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

查询全文数据要微妙的多。我们问的不只是“这个文档匹配查询吗”，而是“该文档匹配查询的程度有多大？”换句话说，该文档与给定查询的相关性如何？

我们很少对全文类型的域做精确匹配。相反，我们希望在文本类型的域中搜索。不仅如此，我们还希望搜索能够理解我们的意图：

- 搜索 UK，会返回包含 United Kingdom 的文档。
- 搜索 jump，会匹配 jumped，jumps，jumping，甚至是 leap。
- 搜索 johnny walker 会匹配 Johnnie Walker，johnnie depp 应该匹配 Johnny Depp。

fox news hunting 应该返回福克斯新闻（ Fox News ）中关于狩猎的故事，同时， fox hunting news 应该返回关于猎狐的故事。为了促进这类在全文域中的查询， Elasticsearch 首先 分析 文档，之后根据结果创建 倒排索引。在接下来的两节，我们会讨论倒排索引和分析过程。

Copyright © Songbai Yang all right reserved, powered by Gitbook 该文件修订时间：2021-03-13

- 1. 倒排索引

1. 倒排索引

Elasticsearch 使用一种称为 倒排索引 的结构，它适用于快速的全文搜索。一个倒排索引由文档中所有不重复词的列表构成，对于其中每个词，有一个包含它的文档列表。

例如，假设我们有两个文档，每个文档的 `content` 域包含如下内容：

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，我们首先将每个文档的 `content` 域拆分成单独的词（我们称它为 词条 或 `tokens`），创建一个包含所有不重复词条的排序列表，然后列出每个词条出现在哪个文档。结果如下所示：

Term	Doc_1	Doc_2
Quick		X
The	X	
brown	X	X
dog	X	
dogs		X
fox	X	
foxes		X
in		X
jumped	X	
lazy	X	X
leap		X
over	X	X
quick	X	
summer		X
the	X	

现在，如果我们想搜索 `quick brown`，我们只需要查找包含每个词条的文档：

Term	Doc_1	Doc_2
brown	X	X
quick	X	
Total	2	1

两个文档都匹配，但是第一个文档比第二个匹配度更高。如果我们使用仅计算匹配词条数量的简单相似性算法，那么，我们可以说，对于我们查询的相关性来讲，第一个文档比第二个文档更佳。

但是，我们目前的倒排索引有一些问题：

- Quick 和 quick 以独立的词条出现，然而用户可能认为它们是相同的词。
- fox 和 foxes 非常相似，就像 dog 和 dogs；他们有相同的词根。
- jumped 和 leap，尽管没有相同的词根，但他们的意思很相近。他们是同义词。

使用前面的索引搜索 +Quick +fox 不会得到任何匹配文档。（记住，+ 前缀表明这个词必须存在。）只有同时出现 Quick 和 fox 的文档才满足这个查询条件，但是第一个文档包含 quick fox，第二个文档包含 Quick foxes。

我们的用户可以合理的期望两个文档与查询匹配。我们可以做的更好。

如果我们将词条规范为标准模式，那么我们可以找到与用户搜索的词条不完全一致，但具有足够相关性的文档。例如：

- Quick 可以小写化为 quick。
- foxes 可以词干提取--变为词根的格式--为 fox。类似的，dogs 可以为提取为 dog。
- jumped 和 leap 是同义词，可以索引为相同的单词 jump。

现在索引看上去像这样：

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

这还远远不够。我们搜索 +Quick +fox 仍然会失败，因为在我们的索引中，已经没有 Quick 了。但是，如果我们对搜索的字符串使用与 content 域相同的标准规则，会变成查询 +quick +fox，这样两个文档都会匹配！

NOTE

这非常重要。你只能搜索在索引中出现的词条，所以索引文本和查询字符串必须标准化为相同的格式。

分词和标准化的过程称为 分析，我们在下个章节讨论。

安装运行elasticsearch

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-13

- 1. 分析与分析器
 - 1.1. 内置分析器
 - 1.2. 什么时候使用分词器
 - 1.3. 测试分词器
 - 1.4. 指定分词器

1. 分析与分析器

分析 包含下面的过程：

- 首先，将一块文本分成适合于倒排索引的独立的 词条，
- 之后，将这些词条统一化为标准格式以提高它们的“可搜索性”，或者 recall

分析器执行上面的工作。分析器 实际上是将三个功能封装到了一个包里：

字符过滤器

首先，字符串按顺序通过每个 字符过滤器。他们的任务是在分词前整理字符串。一个字符过滤器可以用来去掉 HTML，或者将 & 转化成 and。分词器

其次，字符串被 分词器 分为单个的词条。一个简单的分词器遇到空格和标点的时候，可能会将文本拆分成词条。 Token 过滤器 最后，词条按顺序通过每个 token 过滤器。这个过程可能会改变词条（例如，小写化 Quick），删除词条（例如，像 a， and， the 等无用词），或者增加词条（例如，像 jump 和 leap 这种同义词）。

Elasticsearch提供了开箱即用的字符过滤器、分词器和token 过滤器。这些可以组合起来形成自定义的分析器以用于不同的目的。我们会在 自定义分析器 章节详细讨论。

1.1. 内置分析器

但是， Elasticsearch还附带了可以直接使用的预包装的分析器。接下来我们会列出最重要的分析器。为了证明它们的差异，我们看看每个分析器会从下面的字符串得到哪些词条：

```
"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。它是分析各种语言文本最常用的选择。它根据 Unicode 联盟 定义的 单词边界 划分文本。删除绝大部分标点。最后，将词条小写。它会产生

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

简单分析器

简单分析器在任何不是字母的地方分隔文本，将词条小写。它会产生

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

空格分析器

空格分析器在空格的地方划分文本。它会产生

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

语言分析器

特定语言分析器可用于很多语言。它们可以考虑指定语言的特点。例如，英语分析器附带了一组英语无用词（常用单词，例如 `and` 或者 `the`，它们对相关性没有多少影响），它们会被删除。由于理解英语语法的规则，这个分词器可以提取英语单词的词干。

英语 分词器会产生下面的词条：

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

注意看 `transparent`、`calling` 和 `set_trans` 已经变为词根格式。

1.2. 什么时候使用分词器

当我们 索引 一个文档，它的全文域被分析成词条以用来创建倒排索引。但是，当我们在全文域 搜索 的时候，我们需要将查询字符串通过相同的分析过程，以保证我们搜索的词条格式与索引中的词条格式一致。

全文查询，理解每个域是如何定义的，因此它们可以做正确的事：

- 当你查询一个 `全文` 域时，会对查询字符串应用相同的分析器，以产生正确的搜索词条列表。
- 当你查询一个 `精确值` 域时，不会分析查询字符串，而是搜索你指定的精确值。

现在你可以理解在 开始章节 的查询为什么返回那样的结果：

- `date` 域包含一个精确值：单独的词条 `2014-09-15`。
- `_all` 域是一个全文域，所以分词进程将日期转化为三个词条：`2014`，`09`，和 `15`。

当我们在 `_all` 域查询 `2014`，它匹配所有的12条推文，因为它们都含有 `2014`：

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

当我们在 `_all` 域查询 `2014-09-15`，它首先分析查询字符串，产生匹配 `2014`，`09`，或 `15` 中任意词条的查询。这也会匹配所有12条推文，因为它们都含有 `2014`：

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

当我们在 `date` 域查询 `2014-09-15`，它寻找 精确 日期，只找到一个推文：

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

当我们在 `date` 域查询 `2014`，它找不到任何文档，因为没有文档含有这个精确日志：

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

1.3. 测试分词器

有些时候很难理解分词的过程和实际被存储到索引中的词条，特别是你刚接触 Elasticsearch。为了理解发生了什么，你可以使用 `analyze API` 来看文本是如何被分析的。在消息体里，指定分析器和要分析的文本：

```
GET /_analyze
{
  "analyzer": "standard",
  "text": "Text to analyze"
}
```

结果中每个元素代表一个单独的词条：

```
{
  "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` 是实际存储到索引中的词条。`position` 指明词条在原始文本中出现的位置。`start_offset` 和 `end_offset` 指明字符在原始字符串中的位置。

NOTE

每个分析器的 `type` 值都不一样，可以忽略它们。它们在 Elasticsearch 中的唯一作用在于 [keep_types token](#) 过滤器。

`analyze API` 是一个有用的工具，它有助于我们理解 Elasticsearch 索引内部发生了什么，随着深入，我们会进一步讨论它。

1.4. 指定分词器

当 Elasticsearch 在你的文档中检测到一个新的字符串域，它会自动设置其为一个全文 `字符串` 域，使用 `标准` 分析器对它进行分析。

你不希望总是这样。可能你想使用一个不同的分析器，适用于你的数据使用的语言。有时候你想要一个字符串域就是一个字符串域—不使用分析，直接索引你传入的精确值，例如用户ID或者一个内部的状态域或标签。

要做到这一点，我们必须手动指定这些域的映射。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-14

- [1. 映射](#)
 - [1.1. 核心简单域类型](#)
 - [1.2. 查看映射](#)
 - [1.3. 自定义域映射](#)
 - [1.4. index](#)
 - [1.5. analyzer](#)
 - [1.6. 更新映射](#)
 - [1.7. 测试映射](#)

1. 映射

为了能够将时间域视为时间，数字域视为数字，字符串域视为全文或精确值字符串，Elasticsearch 需要知道每个域中数据的类型。这个信息包含在映射中。

如 [数据输入和输出](#) 中解释的，索引中每个文档都有 [类型](#)。每种类型都有它自己的 [映射](#)，或者 [模式定义](#)。映射定义了类型中的域，每个域的数据类型，以及 Elasticsearch 如何处理这些域。映射也用于配置与类型有关的元数据。

我们会在 [类型和映射](#) 详细讨论映射。本节，我们只讨论足够让你入门的内容。

1.1. 核心简单域类型

Elasticsearch 支持如下简单域类型：

- 字符串: `string`
- 整数 : `byte`, `short`, `integer`, `long`
- 浮点数: `float`, `double`
- 布尔型: `boolean`
- 日期: `date`

当你索引一个包含新域的文档—之前未曾出现-- Elasticsearch 会使用 [动态映射](#)，通过 JSON 中基本数据类型，尝试猜测域类型，使用如下规则：

JSON type	域 type
布尔型: <code>true</code> 或者 <code>false</code>	<code>boolean</code>
整数: <code>123</code>	<code>long</code>
浮点数: <code>123.45</code>	<code>double</code>
字符串, 有效日期: <code>2014-09-15</code>	<code>date</code>
字符串: <code>foo bar</code>	<code>string</code>

NOTE

这意味着如果你通过引号(`"123"`)索引一个数字，它会被映射为 `string` 类型，而不是 `long`。但是，如果这个域已经映射为 `long`，那么 Elasticsearch 会尝试将这个字符串转化为 `long`，如果无法转化，则抛出一个异常。

1.2. 查看映射

通过 `_mapping`，我们可以查看 Elasticsearch 在一个或多个索引中的一个或多个类型的映射。在开始章节，我们已经取得索引 `gb` 中类型 `tweet` 的映射：

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

Elasticsearch 根据我们索引的文档，为域(称为 属性)动态生成的映射。

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

Tip

错误的映射，例如 将 `age` 域映射为 `string` 类型，而不是 `integer`，会导致查询出现令人困惑的结果。

检查一下！而不是假设你的映射是正确的。

1.3. 自定义域映射

尽管在很多情况下基本域数据类型已经够用，但你经常需要为单独域自定义映射，特别是字符串域。自定义映射允许你执行下面的操作：

- 全文字符串域和精确值字符串域的区别
- 使用特定语言分析器
- 优化域以适应部分匹配
- 指定自定义数据格式
- 还有更多

域最重要的属性是 `type`。对于不是 `string` 的域，你一般只需要设置

```
type :
```

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

默认，`string` 类型域会被认为包含全文。就是说，它们的值在索引前，会通过一个分析器，针对于这个域的查询在搜索前也会经过一个分析器。

`string` 域映射的两个最重要属性是 `index` 和 `analyzer`。

1.4. index

`index` 属性控制怎样索引字符串。它可以是下面三个值：

analyzed

首先分析字符串，然后索引它。换句话说，以全文索引这个域。

not_analyzed

索引这个域，所以它能够被搜索，但索引的是精确值。不会对它进行分析。

no

不索引这个域。这个域不会被搜索到。

`string` 域 `index` 属性默认是 `analyzed`。如果我们想映射这个字段为一个精确值，我们需要设置它为 `not_analyzed`：

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

其他简单类型（例如 `long`，`double`，`date` 等）也接受 `index` 参数，但有意义的值只有 `no` 和 `not_analyzed`，因为它们永远不会被分析。

1.5. analyzer

对于 `analyzed` 字符串域，用 `analyzer` 属性指定在搜索和索引时使用的分析器。默认，Elasticsearch 使用 `standard` 分析器，但你可以指定一个内置的分析器替代它，例如 `whitespace`、`simple` 和 `english`：

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

在自定义分析器，我们会展示怎样定义和使用自定义分析器。

1.6. 更新映射

当你首次创建一个索引的时候，可以指定类型的映射。你也可以使用 `/_mapping` 为新类型（或者为存在的类型更新映射）增加映射。

>

尽管你可以增加一个存在的映射，你不能修改存在的域映射。如果一个域的映射已经存在，那么该域的数据可能已经被索引。如果你意图修改这个域的映射，索引的数据可能会出错，不能被正常的搜索。

我们可以更新一个映射来添加一个新域，但不能将一个存在的域从 `analyzed` 改为 `not_analyzed`。

为了描述指定映射的两种方式，我们先删除 gb 索引：

```
DELETE /gb
```

然后创建一个新索引，指定 `tweet` 域使用 `english` 分析器：

```
PUT /gb ( a )
{
  "mappings": {
    "tweet" : {
      "properties" : {
        "tweet" : {
          "type" : "string",
          "analyzer": "english"
        },
        "date" : {
          "type" : "date"
        },
        "name" : {
          "type" : "string"
        },
        "user_id" : {
          "type" : "long"
        }
      }
    }
  }
}
```

(a) 通过消息体中指定的 `mappings` 创建了索引。

稍后，我们决定在 `tweet` 映射增加一个新的名为 `tag` 的 `not_analyzed` 的文本域，使用 `_mapping`：

```
PUT /gb/_mapping/tweet
{
  "properties" : {
    "tag" : {
      "type" : "string",
      "index": "not_analyzed"
    }
  }
}
```

注意，我们不需要再次列出所有已存在的域，因为无论如何我们都无法改变它们。新域已经被合并到存在的映射中。

1.7. 测试映射

你可以使用 analyze API 测试字符串域的映射。比较下面两个请求的输出：

```
GET /gb/_analyze
{
  "field": "tweet",
  "text": "Black-cats" ( a)
}

GET /gb/_analyze
{
  "field": "tag",
  "text": "Black-cats" ( a)
}
```

(a) 消息体里面传输我们想要分析的文本。

`tweet` 域产生两个词条 `black` 和 `cat`，`tag` 域产生单独的词条 `Black-cats`。换句话说，我们的映射正常工作。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-14

- **1. 复杂核心域类型**
 - **1.1. 多值域**
 - **1.2. 空域**
 - **1.3. 多层级对象**
 - **1.4. 内部对象的映射**
 - **1.5. 内部对象是如何索引的**
 - **1.6. 内部对象是如何被索引的**

1. 复杂核心域类型

除了我们提到的简单标量数据类型，`JSON` 还有 `null` 值，数组，和对象，这些 Elasticsearch 都是支持的。

1.1. 多值域

很有可能，我们希望 `tag` 域包含多个标签。我们可以以数组的形式索引标签：

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

对于数组，没有特殊的映射需求。任何域都可以包含0、1或者多个值，就像全文域分析得到多个词条。

这暗示 数组中所有的值必须是相同数据类型的。你不能将日期和字符串混在一起。如果你通过索引数组来创建新的域，Elasticsearch 会用数组中第一个值的数据类型作为这个域的 **类型**。

NOTE

当你从 `Elasticsearch` 得到一个文档，每个数组的顺序和你当初索引文档时一样。你得到的 `_source` 域，包含与你索引的一模一样的 `JSON` 文档。

但是，数组是以多值域 索引的—可以搜索，但是无序的。在搜索的时候，你不能指定“第一个”或者“最后一个”。更确切的说，把数组想象成 装在袋子里的值。

1.2. 空域

当然，数组可以为空。这相当于存在零值。事实上，在 `Lucene` 中是不能存储 `null` 值的，所以我们认为存在 `null` 值的域为空域。

下面三种域被认为是空的，它们将不会被索引：

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

1.3. 多层级对象

我们讨论的最后一个 JSON 原生数据类是 对象 -- 在其他语言中称为哈希，哈希 map，字典或者关联数组。

内部对象 经常用于嵌入一个实体或对象到其它对象中。例如，与其在 tweet 文档中包含 user_name 和 user_id 域，我们也可以这样写：

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

1.4. 内部对象的映射

Elasticsearch 会动态监测新的对象域并映射它们为 对象，在 properties 属性 下列出内部域：

```
{
  "gb": {
    "tweet": { ( a)
      "properties": {
        "tweet": { "type": "string" },
        "user": { ( b)
          "type": "object",
          "properties": {
            "id": { "type": "string" },
            "gender": { "type": "string" },
            "age": { "type": "long" },
            "name": { ( b)
              "type": "object",
              "properties": {
                "full": { "type": "string" },
                "first": { "type": "string" },
                "last": { "type": "string" }
              }
            }
          }
        }
      }
    }
  }
}
```

(a) 根对象

(b) 内部对象

user 和 name 域的映射结构与 tweet 类型的相同。事实上， type 映射只是一种特殊的 对象 映射，我们称之为 根对象 。除了它有一些文档元数据的特殊顶级域，例如 _source 和 _all 域，它和其他对象一样。

1.5. 内部对象是如何索引的

Lucene 不理解内部对象。Lucene 文档是由一组键值对列表组成的。为了能让 Elasticsearch 有效地索引内部类，它把我们的文档转化成这样：

```
{
  "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"
}
```

内部域可以通过名称引用（例如，`first`）。为了区分同名的两个域，我们可以使用全路径（例如，`user.name.first`）或 type 名加路径（`tweet.user.name.first`）。

Note

在前面简单扁平的文档中，没有 `user` 和 `user.name` 域。Lucene 索引只有标量和简单值，没有复杂数据结构。

1.6. 内部对象是如何被索引的

最后，考虑包含内部对象的数组是如何被索引的。假设我们有个 `followers` 数组：

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

这个文档会像我们之前描述的那样被扁平化处理，结果如下所示：

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

`{age: 35}` 和 `{name: Mary White}` 之间的相关性已经丢失了，因为每个多值域只是一包无序的值，而不是有序数组。这足以让我们问，“有一个26岁的追随者？”

但是我们不能得到一个准确的答案：“是否有一个26岁名字叫 Alex Jones 的追随者？”

相关内部对象被称为 `nested` 对象，可以回答上面的查询，我们稍后会在嵌套对象中介绍它。

安装运行elasticsearch

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-03-14

- 1. 请求体查询

1. 请求体查询

简易查询—query-string search—对于用命令行进行即席查询（ad-hoc）是非常有用的。然而，为了充分利用查询的强大功能，你应该使用请求体 search API，之所以称之为请求体查询(Full-Body Search)，因为大部分参数是通过Http请求体而非查询字符串来传递的。

请求体查询—下文简称“查询”—不仅可以处理自身的查询请求，还允许你对结果进行片段强调（高亮）、对所有或部分结果进行聚合分析，同时还可以给出你是不是想找的建议，这些建议可以引导使用者快速找到他想要的结果。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-14

- 1. 空查询

1. 空查询

让我们以最简单的 search API 的形式开启我们的旅程，空查询将返回所有索引库 (indices)中的所有文档：

```
GET /_search
{ } ( a)
```

(a) 这是一个空的请求体。

只用一个查询字符串，你就可以在一个、多个或者 `_all` 索引库 (`indices`) 中查询：

```
GET /index_2014*/type/_search
{ }
```

同时你可以使用 `from` 和 `size` 参数来分页：

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

一个带请求体的 GET 请求？

某些特定语言（特别是 JavaScript）的 HTTP 库是不允许 GET 请求带有请求体的。事实上，一些使用者对于 GET 请求可以带请求体感到非常的吃惊。

而事实是这个RFC文档 RFC 7231—一个专门负责处理 HTTP 语义和内容的文档—并没有规定一个带有请求体的 GET 请求应该如何处理！结果是，一些 HTTP 服务器允许这样子，而有一些—特别是那些用于缓存和代理的服务器—则不允许。

对于一个查询请求，Elasticsearch 的工程师偏向于使用 GET 方式，因为他们觉得它比 `POST` 能更好的描述信息检索（`retrieving information`）的行为。然而，因为带请求体的 `GET` 请求并不被广泛支持，所以 search API 同时支持 `POST` 请求：

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

类似的规则可以应用于任何需要带请求体的 GET API。

我们将在聚合 `聚合` 章节深入介绍聚合 (`aggregations`)，而现在，我们将聚焦在查询。

相对于使用晦涩难懂的查询字符串的方式，一个带请求体的查询允许我们使用查询领域特定语言（query domain-specific language）或者Query DSL来写查询语句。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-03-14

- [1. 查询表达式](#)
 - [1.1. 查询语句的结构](#)
 - [1.2. 合并查询语句](#)

1. 查询表达式

查询表达式(`Query DSL`)是一种非常灵活又富有表现力的 查询语言。
Elasticsearch 使用它可以以简单的 `JSON` 接口来展现 `Lucene` 功能的绝大部分。在你的应用中，你应该用它来编写你的查询语句。它可以使你的查询语句更灵活、更精确、易读和易调试。

要使用这种查询表达式，只需将查询语句传递给 `query` 参数：

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

空查询 (`empty search`) `-{}-` 在功能上等价于使用 `match_all` 查询，正如其名字一样，匹配所有文档：

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

1.1. 查询语句的结构

一个查询语句的典型结构：

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

  }
}
```

如果是针对某个字段，那么它的结构如下：

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

    }
  }
}
```

举个例子，你可以使用 `match` 查询语句 来查询 `tweet` 字段中包含 `elasticsearch` 的 `tweet`：

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

完整的查询请求如下：

```
GET /_search  
{  
  "query": {  
    "match": {  
      "tweet": "elasticsearch"  
    }  
  }  
}
```

1.2. 合并查询语句

查询语句(`Query clauses`)就像一些简单的组合块，这些组合块可以彼此之间合并组成更复杂的查询。这些语句可以是如下形式：

- 叶子语句 (`Leaf clauses`) (就像 `match` 语句) 被用于将查询字符串和一个字段 (或者多个字段) 对比。
- 复合(`Compound`)语句主要用于合并其它查询语句。比如，一个 `bool` 语句允许在你需要的时候组合其它语句，无论是 `must` 匹配、`must_not` 匹配还是 `should` 匹配，同时它可以包含不评分的过滤器 (`filters`)：

```
{  
  "bool": {  
    "must": { "match": { "tweet": "elasticsearch" }},  
    "must_not": { "match": { "name": "mary" }},  
    "should": { "match": { "tweet": "full text" }},  
    "filter": { "range": { "age": { "gt": 30 } } }  
  }  
}
```

一条复合语句可以合并任何其它查询语句，包括复合语句，了解这一点是很重要的。这就意味着，复合语句之间可以互相嵌套，可以表达非常复杂的逻辑。

例如，以下查询是为了找出信件正文包含 `business opportunity` 的星标邮件，或者在收件箱正文包含 `business opportunity` 的非垃圾邮件：

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

到目前为止，你不必太在意这个例子的细节，我们会在后面详细解释。最重要的是你要理解到，一条复合语句可以将多条语句 – 叶子语句和其它复合语句 – 合并成一个单一的查询语句。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-14

- [1. 查询与过滤](#)
 - [1.1. 性能差异](#)
 - [1.2. 如何选择查询与过滤](#)

1. 查询与过滤

Elasticsearch 使用的查询语言（DSL）拥有一套查询组件，这些组件可以以无限组合的方式进行搭配。这套组件可以在以下两种情况下使用：过滤情况（`filtering context`）和查询情况（`query context`）。

当使用于过滤情况时，查询被设置成一个“不评分”或者“过滤”查询。即，这个查询只是简单的问一个问题：“这篇文档是否匹配？”回答也是非常的简单，`yes` 或者 `no`，二者必居其一。

- `created` 时间是否在 2013 与 2014 这个区间？
- `status` 字段是否包含 `published` 这个单词？
- `lat_lon` 字段表示的位置是否在指定点的 10km 范围内？

当使用于查询情况时，查询就变成了一个“评分”的查询。和不评分的查询类似，也要去判断这个文档是否匹配，同时它还需要判断这个文档匹配的有多好（匹配程度如何）。此查询的典型用法是用于查找以下文档：

- 查找与 `full text search` 这个词语最佳匹配的文档
- 包含 `run` 这个词，也能匹配 `runs`、`running`、`jog` 或者 `sprint`
- 包含 `quick`、`brown` 和 `fox` 这几个词 — 词之间离的越近，文档相关性越高
- 标有 `lucene`、`search` 或者 `java` 标签 — 标签越多，相关性越高

一个评分查询计算每一个文档与此查询的相关程度，同时将这个相关程度分配给表示相关性的字段 `_score`，并且按照相关性对匹配到的文档进行排序。这种相关性的概念是非常适合全文搜索的情况，因为全文搜索几乎没有完全“正确”的答案。

Note

自 Elasticsearch 问世以来，查询与过滤（queries and filters）就独自成为 Elasticsearch 的组件。但从 Elasticsearch 2.0 开始，过滤（filters）已经从技术上被排除了，同时所有的查询（queries）拥有变成不评分查询的能力。

然而，为了明确和简单，我们用 “filter” 这个词表示不评分、只过滤情况下的查询。你可以把 “filter”、“filtering query” 和 “non-scoring query” 这几个词视为相同的。

相似的，如果单独地不加任何修饰词地使用 “query” 这个词，我们指的是 “scoring query”。

1.1. 性能差异

过滤查询 (Filtering queries) 只是简单的检查包含或者排除，这就使得计算起来非常快。考虑到至少有一个过滤查询 (filtering query) 的结果是“稀少的”(很少匹配的文档)，并且经常使用不评分查询 (non-scoring queries)，结果会被缓存到内存中以便快速读取，所以有各种各样的手段来优化查询结果。

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

多亏倒排索引 (inverted index)，一个简单的评分查询在匹配少量文档时可能与一个涵盖百万文档的filter表现的一样好，甚至会更好。但是在一般情况下，一个filter 会比一个评分的query性能更优异，并且每次都表现的很稳定。

过滤 (filtering) 的目标是减少那些需要通过评分查询 (scoring queries) 进行检查的文档。

1.2. 如何选择查询与过滤

通常的规则是，使用查询 (query) 语句来进行全文搜索或者其它任何需要影响相关性得分的搜索。除此以外的情况都使用过滤 (filters)。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-14

- [1. 最重要的查询](#)
 - [1.1. match_all 查询](#)
 - [1.2. match 查询](#)
 - [1.3. multi_match 查询](#)
 - [1.4. range查询](#)
 - [1.5. term查询](#)
 - [1.6. terms查询](#)
 - [1.7. exists 查询和 missing 查询](#)

1. 最重要的查询

虽然 Elasticsearch 自带了很多的查询，但经常用到的也就那么几个。 我们将在深入搜索章节详细讨论那些查询的细节，接下来我们对最重要的几个查询进行简单介绍。

1.1. match_all 查询

`match_all` 查询简单的匹配所有文档。在没有指定查询方式时，它是默认的查询：

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

它经常与 `filter` 结合使用—例如，检索收件箱里的所有邮件。所有邮件被认为具有相同的相关性，所以都将获得分值为 `1` 的中性 `_score`。

1.2. match 查询

无论你在任何字段上进行的是全文搜索还是精确查询，`match` 查询是你可用的标准查询。

如果你在一个全文字段上使用 `match` 查询，在执行查询前，它将用正确的分析器去分析查询字符串：

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

如果在一个精确值的字段上使用它，例如数字、日期、布尔或者一个 `not_analyzed` 字符串字段，那么它将会精确匹配给定的值：

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

Tip

对于精确值的查询，你可能需要使用 `filter` 语句来取代 `query`，因为 `filter` 将会被缓存。接下来，我们将看到一些关于 `filter` 的例子。

不像我们在 轻量 搜索 章节介绍的字符串查询 (query-string search) ,
match 查询不使用类似 +user_id:2 +tweet:search 的查询语法。它只是去查找给定的单词。这就意味着将查询字段暴露给你的用户是安全的；你需要控制那些允许被查询字段，不易于抛出语法异常。

1.3. multi_match 查询

multi_match 查询可以在多个字段上执行相同的 **match** 查询：

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

1.4. range查询

range 查询找出那些落在指定区间内的数字或者时间：

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

被允许的操作符如下：

gt	大于
gte	大于等于
lt	小于
lte	小于等于

1.5. term查询

term 查询被用于精确值匹配，这些精确值可能是数字、时间、布尔或者那些 **not_analyzed** 的字符串：

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

term 查询对于输入的文本不 分析，所以它将给定的值进行精确查询。

1.6. terms查询

`terms` 查询和 `term` 查询一样，但它允许你指定多值进行匹配。如果这个字段包含了指定值中的任何一个值，那么这个文档满足条件：

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

和 `term` 查询一样，`terms` 查询对于输入的文本不分析。它查询那些精确匹配的值（包括在大小写、重音、空格等方面差异）。

1.7. exists 查询和 missing 查询

`exists` 查询和 `missing` 查询被用于查找那些指定字段中有值 (`exists`) 或无值 (`missing`) 的文档。这与 SQL 中的 `IS NULL` (`missing`) 和 `NOT IS NULL` (`exists`) 在本质上具有共性：

```
{
  "exists": {
    "field": "title"
  }
}
```

这些查询经常用于某个字段有值的情况和某个字段缺值的情况。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-14

- 1. 组合多查询
 - 1.1. 增加带过滤器（filtering）的查询
 - 1.2. constant_score 查询

1. 组合多查询

现实的查询需求从来都没有那么简单；它们需要在多个字段上查询多种多样的文本，并且根据一系列的标准来过滤。为了构建类似的高级查询，你需要一种能够将多查询组合成单一查询的查询方法。

你可以用 `bool` 查询来实现你的需求。这种查询将多查询组合在一起，成为用户自己想要的布尔查询。它接收以下参数：

`must`

文档 必须 匹配这些条件才能被包含进来。

`must_not`

文档 必须 不 匹配这些条件才能被包含进来。

`should`

如果满足这些语句中的任意语句，将增加 `_score`，否则，无任何影响。它们主要用于修正每个文档的相关性得分。

`filter`

必须 匹配，但它以不评分、过滤模式来进行。这些语句对评分没有贡献，只是根据过滤标准来排除或包含文档。

由于这是我们看到的第一个包含多个查询的查询，所以有必要讨论一下相关性得分是如何组合的。每一个子查询都独自地计算文档的相关性得分。一旦他们的得分被计算出来，`bool` 查询就将这些得分进行合并并且返回一个代表整个布尔操作的得分。

下面的查询用于查找 `title` 字段匹配 `how to make millions` 并且不被标识为 `spam` 的文档。那些被标识为 `starred` 或在 `2014` 之后的文档，将比另外那些文档拥有更高的排名。如果 两者 都满足，那么它排名将更高：

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

Tip

如果没有 `must` 语句，那么至少需要能够匹配其中的一条 `should` 语句。但，如果存在至少一条 `must` 语句，则对 `should` 语句的匹配没有要求。

1.1. 增加带过滤器（filtering）的查询

如果我们不想因为文档的时间而影响得分，可以用 `filter` 语句来重写前面的例子：

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

(a) range 查询已经从 `should` 语句中移到 `filter` 语句

通过将 `range` 查询移到 `filter` 语句中，我们将它转成不评分的查询，将不再影响文档的相关性排名。由于它现在是一个不评分的查询，可以使用各种对 `filter` 查询有效的优化手段来提升性能。

所有查询都可以借鉴这种方式。将查询移到 `bool` 查询的 `filter` 语句中，这样它就自动的转成一个不评分的 `filter` 了。

如果你需要通过多个不同的标准来过滤你的文档，`bool` 查询本身也可以被用做不评分的查询。简单地将它放置到 `filter` 语句中并在内部构建布尔逻辑：

```
{
  "bool": {
    "must": { "match": { "title": "how to make millions" }},
    "must_not": { "match": { "tag": "spam" }},
    "should": [
      { "match": { "tag": "starred" }}
    ],
    "filter": {
      "bool": { (a)
        "must": [
          { "range": { "date": { "gte": "2014-01-01" }}},
          { "range": { "price": { "lte": 29.99 }}}
        ],
        "must_not": [
          { "term": { "category": "ebooks" }}
        ]
      }
    }
  }
}
```

(a) 将 `bool` 查询包裹在 `filter` 语句中，我们可以在过滤标准中增加布尔逻辑

通过混合布尔查询，我们可以在我们的查询请求中灵活地编写 `scoring` 和 `filtering` 查询逻辑。

1.2. constant_score 查询

尽管没有 bool 查询使用这么频繁， constant_score 查询也是你工具箱里有用的查询工具。它将一个不变的常量评分应用于所有匹配的文档。它被经常用于你只需要执行一个 filter 而没有其它查询（例如，评分查询）的情况下。

可以使用它来取代只有 filter 语句的 bool 查询。在性能上是完全相同的，但对于提高查询简洁性和清晰度有很大帮助。

```
{  
  "constant_score": {  
    "filter": {  
      "term": { "category": "ebooks" }  ( a)  
    }  
  }  
}
```

(a) term 查询被放置在 constant_score 中，转成不评分的 filter。这种方式可以用来取代只有 filter 语句的 bool 查询。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-03-14

- **1. 验证查询**
 - **1.1. 理解错误信息**
 - **1.2. 理解查询语句**

1. 验证查询

查询可以变得非常的复杂，尤其和不同的分析器与不同的字段映射结合时，理解起来就有点困难了。不过 `validate-query API` 可以用来验证查询是否合法。

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

以上 `validate` 请求的应答告诉我们这个查询是不合法的：

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

1.1. 理解错误信息

为了找出查询不合法的原因，可以将 `explain` 参数加到查询字符串中：

```
GET /gb/tweet/_validate/query?explain ( a )
{
  "query": {
    "tweet" : {
      "match" : "really powerful"
    }
  }
}
```

(a) `explain` 参数可以提供更多关于查询不合法的信息。

很明显，我们将查询类型(`match`)与字段名称(`tweet`)搞混了：

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

1.2. 理解查询语句

对于合法查询，使用 `explain` 参数将返回可读的描述，这对准确理解 Elasticsearch 是如何解析你的 `query` 是非常有用的：

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

我们查询的每一个 `index` 都会返回对应的 `explanation`，因为每一个 `index` 都有自己的映射和分析器：

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

从 `explanation` 中可以看出，匹配 `really powerful` 的 `match` 查询被重写为两个针对 `tweet` 字段的 `single-term` 查询，一个 `single-term` 查询对应查询字符串分出来的一个 `term`。

当然，对于索引 `us`，这两个 `term` 分别是 `really` 和 `powerful`，而对于索引 `gb`，`term` 则分别是 `realli` 和 `power`。之所以出现这个情况，是由于我们将索引 `gb` 中 `tweet` 字段的分析器修改为 `english` 分析器。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-03-14

- 1. 排序与相关性

1. 排序与相关性

默认情况下，返回的结果是按照 `相关性` 进行排序的——最相关的文档排在最前。在本章的后面部分，我们会解释 `相关性` 意味着什么以及它是如何计算的，不过让我们首先看看 `sort` 参数以及如何使用它。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-14

- **1. 排序**
 - **1.1. 按照字段的值排序**
 - **1.2. 多级排序**
 - **1.3. 多值字段的排序**

1. 排序

为了按照相关性来排序，需要将相关性表示为一个数值。在 Elasticsearch 中，相关性得分由一个浮点数进行表示，并在搜索结果中通过 `_score` 参数返回，默认排序是 `_score` 降序。

有时，相关性评分对你来说并没有意义。例如，下面的查询返回所有 `user_id` 字段包含 `1` 的结果：

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

这里没有一个有意义的分数：因为我们使用的是 `filter`（过滤），这表明我们只希望获取匹配 `user_id: 1` 的文档，并没有试图确定这些文档的相关性。实际上文档将按照随机顺序返回，并且每个文档都会评为零分。

Note

如果评分为零对你造成了困扰，你可以使用 `constant_score` 查询进行替代：

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

这将让所有文档应用一个恒定分数（默认为 `1`）。它将执行与前述查询相同的查询，并且所有的文档将像之前一样随机返回，这些文档只是有了一个分数而不是零分。

1.1. 按照字段的值排序

在这个案例中，通过时间来对 `tweets` 进行排序是有意义的，最新的 `tweets` 排在最前。我们可以使用 `sort` 参数进行实现：

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

你会注意到结果中的两个不同点：

```
"hits": {
  "total": 6,
  "max_score": null, (a)
  "hits": [
    {
      "_index": "us",
      "_type": "tweet",
      "_id": "14",
      "_score": null, (a)
      "_source": {
        "date": "2014-09-24",
        ...
      },
      "sort": [ 1411516800000 ] (b)
    },
    ...
  ]
}
```

(a) `_score` 不被计算，因为它并没有用于排序。

(b) `date` 字段的值表示为自 epoch (January 1, 1970 00:00:00 UTC) 以来的毫秒数，通过 `sort` 字段的值进行返回。

首先我们在每个结果中有一个新的名为 `sort` 的元素，它包含了我们用于排序的值。在这个案例中，我们按照 `date` 进行排序，在内部被索引为自 epoch 以来的毫秒数。`long` 类型数 `1411516800000` 等价于日期字符串 `2014-09-24 00:00:00 UTC`。

其次 `_score` 和 `max_score` 字段都是 `null`。计算 `_score` 的花销巨大，通常仅用于排序；我们并不根据相关性排序，所以记录 `_score` 是没有意义的。如果无论如何你都要计算 `_score`，你可以将 `track_scores` 参数设置为 `true`。

Tip

一个简便方法是，你可以指定一个字段用来排序：

`"sort": "number_of_children"` 字段将会默认升序排序，而按照 `_score` 的值进行降序排序。

1.2. 多级排序

假定我们想要结合使用 `date` 和 `_score` 进行查询，并且匹配的结果首先按照日期排序，然后按照相关性排序：

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

排序条件的顺序是很重要的。结果首先按第一个条件排序，仅当结果集的第一个 `sort` 值完全相同时才会按照第二个条件进行排序，以此类推。

多级排序并不一定包含 `_score`。你可以根据一些不同的字段进行排序，如地理距离或是脚本计算的特定值。

Note

`Query-string` 搜索也支持自定义排序，可以在查询字符串中使用 `sort` 参数：

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

1.3. 多值字段的排序

一种情形是字段有多个值的排序，需要记住这些值并没有固有的顺序；一个多值的字段仅仅是多个值的包装，这时应该选择哪个进行排序呢？

对于数字或日期，你可以将多值字段减为单值，这可以通过使用 `min`、`max`、`avg` 或是 `sum` 排序模式。例如你可以按照每个 `date` 字段中的最早日期进行排序，通过以下方法：

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

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-03-14

- 1. 字符串排序与多字段

1. 字符串排序与多字段

被解析的字符串字段也是多值字段，但是很少会按照你想要的方式进行排序。如果你想分析一个字符串，如 `fine old art`，这包含 3 项。我们很可能想要按第一项的字母排序，然后按第二项的字母排序，诸如此类，但是 Elasticsearch 在排序过程中没有这样的信息。

你可以使用 `min` 和 `max` 排序模式（默认是 `min`），但是这会导致排序以 `art` 或是 `old`，任何一个都不是所希望的。

为了以字符串字段进行排序，这个字段应仅包含一项：整个 `not_analyzed` 字符串。但是我们仍需要 `analyzed` 字段，这样才能以全文进行查询

一个简单的方法是用两种方式对同一个字符串进行索引，这将在文档中包括两个字段：`analyzed` 用于搜索，`not_analyzed` 用于排序

但是保存相同的字符串两次在 `_source` 字段是浪费空间的。我们真正想要做的是传递一个单字段但是却用两种方式索引它。所有的 `_core_field` 类型 (`strings`, `numbers`, `Booleans`, `dates`) 接收一个 `fields` 参数

该参数允许你转化一个简单的映射如：

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

为一个多字段映射如：

```
"tweet": { ( a )  
    "type": "string",  
    "analyzer": "english",  
    "fields": {  
        "raw": { ( b )  
            "type": "string",  
            "index": "not_analyzed"  
        }  
    }  
}
```

(a) tweet 主字段与之前的一样：是一个 `analyzed` 全文字段。
(b) 新的 `tweet.raw` 子字段是 `not_analyzed`。

现在，至少只要我们重新索引了我们的数据，使用 `tweet` 字段用于搜索，`tweet.raw` 字段用于排序：



以全文 `analyzed` 字段排序会消耗大量的内存。获取更多信息请看聚合与分析。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-03-14

- 1. 什么是相关性?
 - 1.1. 理解评分标准

1. 什么是相关性?

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，用一个正浮点数字段 `_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生成一个 `_score` 字段。评分的计算方式取决于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，但是通常我们说的 `relevance` 是我们用来计算全文本字段的值相对于全文本检索词相似程度的算法。

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

检索词频率

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。

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` 字段权重更大。

单个查询可以联合使用 `TF/IDF` 和其他方式，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。

1.1. 理解评分标准

当调试一条复杂的查询语句时，想要理解 `_score` 究竟是如何计算是比较困难的。

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。

Copyright © Songbai Yang all right reserved, powered by Gitbook 该文件修订时间：2021-03-26

- 1. 聚合相关知识点
- 2. 聚合

1. 聚合相关知识点

聚合

2. 聚合

在这之前，本书致力于搜索。通过搜索，如果我们有一个查询并且希望找到匹配这个查询的文档集，就好比在大海捞针。

通过聚合，我们会得到一个数据的概览。我们需要的是分析和总结全套的数据而不是寻找单个文档：

- 在大海里有多少针？
- 针的平均长度是多少？
- 按照针的制造商来划分，针的长度中位数是多少？
- 每月加入到海中的针有多少？

聚合也可以回答更加细微的问题：

- 你最受欢迎的针的制造商是什么？
- 这里面有异常的针么？
- 聚合允许我们向数据提出一些复杂的问题。虽然功能完全不同于搜索，但它使用相同的数据结构。这意味着聚合的执行速度很快并且就像搜索一样几乎是实时的。

这对报告和仪表盘是非常强大的。你可以实时显示你的数据，让你立即回应，而不是对你的数据进行汇总（需要一周时间去运行的 Hadoop 任务），您的报告随着你的数据变化而变化，而不是预先计算的、过时的和不相关的。

最后，聚合和搜索是一起的。这意味着你可以在单个请求里同时对相同的数据进行搜索/过滤和分析。并且由于聚合是在用户搜索的上下文里计算的，你不只是显示四星酒店的数量，而是显示匹配查询条件的四星酒店的数量。

聚合是如此强大以至于许多公司已经专门为数据分析建立了大型 Elasticsearch 集群。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-13

- 1. 高阶概念

1. 高阶概念

类似于 DSL 查询表达式，聚合也有 可组合 的语法：独立单元的功能可以被混合起来提供你需要的自定义行为。这意味着只需要学习很少的基本概念，就可以得到几乎无尽的组合。

要掌握聚合，你只需要明白两个主要的概念：

- 桶 (Buckets)

满足特定条件的文档的集合

- 指标 (Metrics)

对桶内的文档进行统计计算

这就是全部了！每个聚合都是一个或者多个桶和零个或者多个指标的组合。翻译成粗略的SQL语句来解释吧：

```
SELECT COUNT(color)
FROM table
GROUP BY color
```

COUNT(color) 相当于指标。

GROUP BY color 相当于桶。

桶在概念上类似于 SQL 的分组 (GROUP BY)，而指标则类似于 COUNT()、SUM()、MAX() 等统计方法。

让我们深入这两个概念 并且了解和这两个概念相关的东西。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-13

- [1. 桶](#)

1. 桶

桶 简单来说就是满足特定条件的文档的集合：

一个雇员属于 男性 桶或者 女性 桶 奥尔巴尼属于 纽约 桶 日期2014-10-28属于 十月 桶 当聚合开始被执行，每个文档里面的值通过计算来决定符合哪个桶的条件。如果匹配到，文档将放入相应的桶并接着进行聚合操作。

桶也可以被嵌套在其他桶里面，提供层次化的或者有条件的划分方案。例如，辛辛那提会被放入俄亥俄州这个桶，而 整个 俄亥俄州桶会被放入美国这个桶。

Elasticsearch 有很多类型的桶，能让你通过很多种方式来划分文档（时间、最受欢迎的词、年龄区间、地理位置等等）。其实根本上都是通过同样的原理进行操作：基于条件来划分文档。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-13

- 1. 指标
- 2. 桶和指标的组合

1. 指标

桶能让我们划分文档到有意义的集合，但是最终我们需要的是对这些桶内的文档进行一些指标的计算。分桶是一种达到目的的手段：它提供了一种给文档分组的方法来让我们可以计算感兴趣的指标。

大多数 指标 是简单的数学运算（例如最小值、平均值、最大值，还有汇总），这些是通过文档的值来计算。

在实践中，指标能让你计算像平均薪资、最高出售价格、95%的查询延迟这样的数据。

2. 桶和指标的组合

聚合 是由桶和指标组成的。聚合可能只有一个桶，可能只有一个指标，或者可能两个都有。也有可能有一些桶嵌套在其他桶里面。

例如，我们可以通过所属国家来划分文档（桶），然后计算每个国家的平均薪酬（指标）。

由于桶可以被嵌套，我们可以实现非常多并且非常复杂的聚合：

- 1.通过国家划分文档（桶）
- 2.然后通过性别划分每个国家（桶）
- 3.然后通过年龄区间划分每种性别（桶）
- 4.最后，为每个年龄区间计算平均薪酬（指标）

最后将告诉你每个 <国家, 性别, 年龄> 组合的平均薪酬。所有的这些都在一个请求内完成并且只遍历一次数据！

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-07-13

- 1. 尝试聚合

1. 尝试聚合

尝试聚合

我们可以用以下几页定义不同的聚合和它们的语法，但学习聚合的最佳途径就是用实例来说明。一旦我们获得了聚合的思想，以及如何合理地嵌套使用它们，那么语法就变得不那么重要了。

聚合的桶操作和度量的完整用法可以在 Elasticsearch 参考 中找到。本章中会涵盖其中很多内容，但在阅读完本章后查看它会有助于我们对它的整体能力有所了解。

所以让我们先看一个例子。我们将会创建一些对汽车经销商有用的聚合，数据是关于汽车交易的信息：车型、制造商、售价、何时被出售等。

首先我们批量索引一些数据：

```
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" }
```

有了数据，开始构建我们的第一个聚合。汽车经销商可能会想知道哪个颜色的汽车销量最好，用聚合可以轻易得到结果，用 terms 桶操作：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "popular_colors" : {
      "terms" : {
        "field" : "color"
      }
    }
  }
}
```

聚合操作被置于顶层参数 aggs 之下（如果你愿意，完整形式 aggregations 同样有效）。然后，可以为聚合指定一个我们想要名称，本例中是： popular_colors 。最后，定义单个桶的类型 terms 。

聚合是在特定搜索结果背景下执行的，这也就是说它只是查询请求的另外一个顶层参数（例如，使用 `/_search` 端点）。]聚合可以与查询结对，但我们会晚些在 [限定聚合的范围 \(Scoping Aggregations\)](#) 中来解决这个问题。

可能会注意到我们将 `size` 设置成 0。我们并不关心搜索结果的具体内容，所以将返回记录数设置为 0 来提高查询速度。设置 `size: 0` 与 Elasticsearch 1.x 中使用 `count` 搜索类型等价。

然后我们为聚合定义一个名字，名字的选择取决于使用者，响应的结果会以我们定义的名字为标签，这样应用就可以解析得到的结果。

随后我们定义聚合本身，在本例中，我们定义了一个单 `terms` 桶。这个 `terms` 桶会为每个碰到的唯一词项动态创建新的桶。因为我们告诉它使用 `color` 字段，所以 `terms` 桶会为每个颜色动态创建新桶。

让我们运行聚合并查看结果：

```
{
...
  "hits": {
    "hits": []
  },
  "aggregations": {
    "popular_colors": {
      "buckets": [
        {
          "key": "red",
          "doc_count": 4
        },
        {
          "key": "blue",
          "doc_count": 2
        },
        {
          "key": "green",
          "doc_count": 2
        }
      ]
    }
  }
}
```

因为我们设置了 `size` 参数，所以不会有 `hits` 搜索结果返回。

`popular_colors` 聚合是作为 `aggregations` 字段的一部分被返回的。

每个桶的 `key` 都与 `color` 字段里找到的唯一词对应。它总会包含 `doc_count` 字段，告诉我们包含该词项的文档数量。

每个桶的数量代表该颜色的文档数量。

响应包含多个桶，每个对应一个唯一颜色（例如：红或绿）。每个桶也包括聚合进该桶的所有文档的数量。例如，有四辆红色的车。

前面的这个例子完全是实时执行的：一旦文档可以被搜到，它就能被聚合。这也就意味着我们可以直接将聚合的结果源源不断的传入图形库，然后生成实时的仪表盘。不久，你又销售了一辆银色的车，我们的图形就会立即动态更新银色车的统计信息。

瞧！这就是我们的第一个聚合！

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-07-13

- 1. 添加度量指标

1. 添加度量指标

前面的例子告诉我们每个桶里面的文档数量，这很有用。但通常，我们的应用需要提供更复杂的文档度量。例如，每种颜色汽车的平均价格是多少？

为了获取更多信息，我们需要告诉 Elasticsearch 使用哪个字段，计算何种度量。这需要将度量 嵌套 在桶内， 度量会基于桶内的文档计算统计结果。

让我们继续为汽车的例子加入 average 平均度量：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs": {
    "colors": {
      "terms": {
        "field": "color"
      },
      "aggs": {
        "avg_price": {
          "avg": {
            "field": "price"
          }
        }
      }
    }
  }
}
```

为度量新增 aggs 层。为度量指定名字： avg_price 。最后，为 price 字段定义 avg 度量。

正如所见，我们用前面的例子加入了新的 aggs 层。这个新的聚合层让我们可以将 avg 度量嵌套置于 terms 桶内。实际上，这就为每个颜色生成了平均价格。

正如 颜色 的例子，我们需要给度量起一个名字（ avg_price ）这样可以稍后根据名字获取它的值。最后，我们指定度量本身（ avg ）以及我们想要计算平均值的字段（ price ）：

```
{  
...  
    "aggregations": {  
        "colors": {  
            "buckets": [  
                {  
                    "key": "red",  
                    "doc_count": 4,  
                    "avg_price": {  
                        "value": 32500  
                    }  
                },  
                {  
                    "key": "blue",  
                    "doc_count": 2,  
                    "avg_price": {  
                        "value": 20000  
                    }  
                },  
                {  
                    "key": "green",  
                    "doc_count": 2,  
                    "avg_price": {  
                        "value": 21000  
                    }  
                }  
            ]  
        }  
    }  
...  
}
```

响应中的新字段 avg_price

尽管响应只发生很小改变，实际上我们获得的数据是增长了。之前，我们知道有四辆红色的车，现在，红色车的平均价格是 \$32, 500 美元。这个信息可以直接显示在报表或者图形中。

Copyright © Songbai Yang all right reserved, powered by Gitbook 该文件修订时间：2021-07-13

- 1. 嵌套桶

1. 嵌套桶

在我们使用不同的嵌套方案时，聚合的力量才能真正得以显现。在前例中，我们已经看到如何将一个度量嵌入桶中，它的功能已经十分强大了。

但真正令人激动的分析来自于将桶嵌套进另外一个桶 所能得到的结果。现在，我们想知道每个颜色的汽车制造商的分布：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs": {
    "colors": {
      "terms": {
        "field": "color"
      },
      "aggs": {
        "avg_price": {
          "avg": {
            "field": "price"
          }
        },
        "make": {
          "terms": {
            "field": "make"
          }
        }
      }
    }
  }
}
```

注意前例中的 avg_price 度量仍然保持原位。

另一个聚合 make 被加入到了 color 颜色桶中。

这个聚合是 terms 桶，它会为每个汽车制造商生成唯一的桶。

这里发生了一些有趣的事。首先，我们可能会观察到之前例子中的 avg_price 度量完全没有变化，还在原来的位置。一个聚合的每个层级都可以有多个度量或桶，avg_price 度量告诉我们每种颜色汽车的平均价格。它与其他的桶和度量相互独立。

这对我们的应用非常重要，因为这里面有很多相互关联，但又完全不同的度量需要收集。聚合使我们能够用一次数据请求获得所有的这些信息。

另外一件值得注意的重要事情是我们新增的这个 make 聚合，它是一个 terms 桶（嵌套在 colors、terms 桶内）。这意味着它会为数据集中的每个唯一组合生成（color、make）元组。

让我们看看返回的响应（为了简单我们只显示部分结果）：

```
{  
...  
    "aggregations": {  
        "colors": {  
            "buckets": [  
                {  
                    "key": "red",  
                    "doc_count": 4,  
                    "make": {  
                        "buckets": [  
                            {  
                                "key": "honda",  
                                "doc_count": 3  
                            },  
                            {  
                                "key": "bmw",  
                                "doc_count": 1  
                            }  
                        ]  
                    },  
                    "avg_price": {  
                        "value": 32500  
                    }  
                },  
                ...  
            ]  
        }  
    }  
}
```

正如期望的那样，新的聚合嵌入在每个颜色桶中。

现在我们看见按不同制造商分解的每种颜色下车辆信息。

最终，我们看到前例中的 avg_price 度量仍然维持不变。

响应结果告诉我们以下几点：

- 红色车有四辆。
- 红色车的平均售价是 \$32, 500 美元。
- 其中三辆是 Honda 本田制造，一辆是 BMW 宝马制造。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-07-13

- 1. 最后的修改

1. 最后的修改

让我们回到话题的原点，在进入新话题之前，对我们的示例做最后一个修改，为每个汽车生成商计算最低和最高的价格：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "aggs": {
        "colors": {
            "terms": {
                "field": "color"
            },
            "aggs": {
                "avg_price": { "avg": { "field": "price" } },
                "make" : {
                    "terms" : {
                        "field" : "make"
                    },
                    "aggs" : {
                        "min_price" : { "min": { "field": "price" } },
                        "max_price" : { "max": { "field": "price" } }
                    }
                }
            }
        }
    }
}
```

我们需要增加另外一个嵌套的 aggs 层级。

然后包括 min 最小度量。

以及 max 最大度量。

得到以下输出（只显示部分结果）：

```
{  
...  
    "aggregations": {  
        "colors": {  
            "buckets": [  
                {  
                    "key": "red",  
                    "doc_count": 4,  
                    "make": {  
                        "buckets": [  
                            {  
                                "key": "honda",  
                                "doc_count": 3,  
                                "min_price": {  
                                    "value": 10000  
                                },  
                                "max_price": {  
                                    "value": 20000  
                                }  
                            },  
                            {  
                                "key": "bmw",  
                                "doc_count": 1,  
                                "min_price": {  
                                    "value": 80000  
                                },  
                                "max_price": {  
                                    "value": 80000  
                                }  
                            }  
                        ]  
                    },  
                    "avg_price": {  
                        "value": 32500  
                    }  
                },  
                ...  
            ]  
        }  
    }  
}
```

min 和 max 度量现在出现在每个汽车制造商（make）下面。

有了这两个桶，我们可以对查询的结果进行扩展并得到以下信息：

- 有四辆红色车。
- 红色车的平均售价是 \$32, 500 美元。
- 其中三辆红色车是 Honda 本田制造，一辆是 BMW 宝马制造。
- 最便宜的红色本田售价为 \$10, 000 美元。
- 最贵的红色本田售价为 \$20, 000 美元。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-07-13

- 1. 按时间统计

1. 按时间统计

如果搜索是在 Elasticsearch 中使用频率最高的，那么构建按时间统计的 date_histogram 紧随其后。为什么你会想用 date_histogram 呢？

假设你的数据带时间戳。无论是什么数据（Apache 事件日志、股票买卖交易时间、棒球运动时间）只要带有时间戳都可以进行 date_histogram 分析。当你的数据有时间戳，你总是想在 时间 维度上构建指标分析：

- 今年每月销售多少台汽车？
- 这只股票最近 12 小时的价格是多少？
- 我们网站上周每小时的平均响应延迟时间是多少？

虽然通常的 histogram 都是条形图，但 date_histogram 倾向于转换成线状图以展示时间序列。许多公司用 Elasticsearch 仅仅 只是为了分析时间序列数据。date_histogram 分析是它们最基本的需要。

date_histogram 与 通常的 histogram 类似。但不是在代表数值范围的数值字段上构建 buckets，而是在时间范围内构建 buckets。因此每一个 bucket 都被定义成一个特定的日期大小(比如，1个月或2.5天)。

可以用通常的 histogram 进行时间分析吗？

从技术上来讲，是可以的。通常的 histogram bucket（桶）是可以处理日期的。但是它不能自动识别日期。而用 date_histogram，你可以指定时间段如1个月，它能聪明地知道2月的天数比12月少。

date_histogram 还具有另外一个优势，即能合理地处理时区，这可以使你用客户端的时区进行图标定制，而不是用服务器端时区。通常的 histogram 会把日期看做是数字，这意味着你必须以微秒为单位指明时间间隔。另外聚合并不知道日历时间间隔，使得它对于日期而言几乎没什么用处。

我们的第一个例子将构建一个简单的折线图来回答如下问题：每月销售多少台汽车？

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs": {
    "sales": {
      "date_histogram": {
        "field": "sold",
        "interval": "month",
        "format": "yyyy-MM-dd"
      }
    }
  }
}
```

时间间隔要求是日历术语(如每个 bucket 1 个月)。

我们提供日期格式以便 buckets 的键值便于阅读。

我们的查询只有一个聚合，每月构建一个 bucket。这样我们可以得到每个月销售的汽车数量。另外还提供了一个额外的 format 参数以便 buckets 有 "好看的" 键值。然而在内部，日期仍然是被简单表示成数值。这可能会使得 UI 设计者抱怨，因此可以提供常用的日期格式进行格式化以更方便阅读。

结果既符合预期又有一点出人意料（看看你是否能找到意外之处）：

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

聚合结果已经完全展示了。正如你所见，我们有代表月份的 buckets，每个月的文档数目，以及美化后的 key_as_string。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-07-13

- 1. 范围限定的聚合
 - 1.1. 全局桶

1. 范围限定的聚合

所有聚合的例子到目前为止，你可能已经注意到，我们的搜索请求省略了一个 query。整个请求只不过是一个聚合。

聚合可以与搜索请求同时执行，但是我们需要理解一个新概念：范围。默认情况下，聚合与查询是对同一范围进行操作的，也就是说，聚合是基于我们查询匹配的文档集合进行计算的。

让我们看看第一个聚合的示例：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "colors" : {
      "terms" : {
        "field" : "color"
      }
    }
  }
}
```

我们可以看到聚合是隔离的。现实中，Elasticsearch 认为 "没有指定查询" 和 "查询所有文档" 是等价的。前面这个查询内部会转化成下面的这个请求：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "query" : {
    "match_all" : {}
  },
  "aggs" : {
    "colors" : {
      "terms" : {
        "field" : "color"
      }
    }
  }
}
```

因为聚合总是对查询范围内的结果进行操作的，所以一个隔离的聚合实际上是在对 match_all 的结果范围操作，即所有的文档。

一旦有了范围的概念，我们就能更进一步对聚合进行自定义。我们前面所有的示例都是对所有数据计算统计信息的：销量最高的汽车，所有汽车的平均售价，最佳销售月份等等。

利用范围，我们可以问“福特在售车有多少种颜色？”诸如此类的问题。可以简单的在请求中加上一个查询（本例中为 match 查询）：

```
GET /cars/transactions/_search
{
  "query": {
    "match": {
      "make": "ford"
    }
  },
  "aggs": {
    "colors": {
      "terms": {
        "field": "color"
      }
    }
  }
}
```

因为我们没有指定 "size" : 0 , 所以搜索结果和聚合结果都被返回了：

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

看上去这并没有什么，但却对高大上的仪表盘来说至关重要。加入一个搜索栏可以将任何静态的仪表板变成一个实时数据搜索设备。这让用户可以搜索数据，查看所有实时更新的图形（由于聚合的支持以及对查询范围的限定）。这是 Hadoop 无

法做到的！

1.1. 全局桶

通常我们希望聚合是在查询范围内的，但有时我们也想要搜索它的子集，而聚合的对象却是所有数据。

例如，比方说我们知道想知道福特汽车与所有汽车平均售价的比较。我们可以用普通的聚合（查询范围内的）得到第一个信息，然后用全局桶获得第二个信息。

全局桶包含所有的文档，它无视查询的范围。因为它还是一个桶，我们可以像平常一样将聚合嵌套在内：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "query" : {
    "match" : {
      "make" : "ford"
    }
  },
  "aggs" : {
    "single_avg_price": {
      "avg" : { "field" : "price" }
    },
    "all": {
      "global" : {},
      "aggs" : {
        "avg_price": {
          "avg" : { "field" : "price" }
        }
      }
    }
  }
}
```

聚合操作在查询范围内（例如：所有文档匹配 ford） global 全局桶没有参数。聚合操作针对所有文档，忽略汽车品牌。

single_avg_price 度量计算是基于查询范围内所有文档，即所有福特汽车。
avg_price 度量是嵌套在全局桶下的，这意味着它完全忽略了范围并对所有文档进行计算。聚合返回的平均值是所有汽车的平均售价。

如果能一直坚持读到这里，应该知道我们有个真言：尽可能的使用过滤器。它同样可以应用于聚合，在下一章中，我们会展示如何对聚合结果进行过滤而不是仅对查询范围做限定。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-07-14

- [1. 过滤和聚合](#)

1. 过滤和聚合

聚合范围限定还有一个自然的扩展就是过滤。因为聚合是在查询结果范围内操作的，任何可以适用于查询的过滤器也可以应用在聚合上。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-14

- 1. 过滤

1. 过滤

如果我们想找到售价在 \$10,000 美元之上的所有汽车同时也为这些车计算平均售价，可以简单地使用一个 `constant_score` 查询和 `filter` 约束：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "query" : {
        "constant_score": {
            "filter": {
                "range": {
                    "price": {
                        "gte": 10000
                    }
                }
            }
        },
        "aggs" : {
            "single_avg_price": {
                "avg" : { "field" : "price" }
            }
        }
    }
}
```

这正如我们在前面章节中讨论过那样，从根本上讲，使用 `non-scoring` 查询和使用 `match` 查询没有任何区别。查询（包括了一个过滤器）返回一组文档的子集，聚合正是操作这些文档。使用 `filtering query` 会忽略评分，并有可能会缓存结果数据等等。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-14

- 1. 过滤痛

1. 过滤痛

但是如果我们只想对聚合结果过滤怎么办？假设我们正在为汽车经销商创建一个搜索页面，我们希望显示用户搜索的结果，但是我们同时也想在页面上提供更丰富的信息，包括（与搜索匹配的）上个月度汽车的平均售价。

这里我们无法简单的做范围限定，因为有两个不同的条件。搜索结果必须是 `ford`，但是聚合结果必须满足 `ford AND sold > now - 1M`。

为了解决这个问题，我们可以用一种特殊的桶，叫做 `filter`（注：过滤桶）。我们可以指定一个过滤桶，当文档满足过滤桶的条件时，我们将其加入到桶内。

查询结果如下：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "query": {
    "match": {
      "make": "ford"
    }
  },
  "aggs": {
    "recent_sales": {
      "filter": {
        "range": {
          "sold": {
            "from": "now-1M"
          }
        }
      },
      "aggs": {
        "average_price": {
          "avg": {
            "field": "price"
          }
        }
      }
    }
  }
}
```

使用 `filter` 桶在查询范围基础上应用过滤器。

`avg` 度量只会对 `ford` 和上个月售出的文档计算平均售价。

因为 `filter` 桶和其他桶的操作方式一样，所以可以随意将其他桶和度量嵌入其中。所有嵌套的组件都会“继承”这个过滤，这使我们可以按需针对聚合过滤出选择部分。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-07-14

- 1. 后过滤器

1. 后过滤器

目前为止，我们可以同时对搜索结果和聚合结果进行过滤（不计算得分的 filter 查询），以及针对聚合结果的一部分进行过滤（filter 桶）。

我们可能会想，“只过滤搜索结果，不过滤聚合结果呢？”答案是使用 post_filter。

它是接收一个过滤器的顶层搜索请求元素。这个过滤器在查询之后执行（这正是该过滤器的名字的由来：它在查询之后 post 执行）。正因为它在查询之后执行，它对查询范围没有任何影响，所以对聚合也不会有任何影响。

我们可以利用这个行为对查询条件应用更多的过滤器，而不会影响其他的操作，就如 UI 上的各个分类面。让我们为汽车经销商设计另外一个搜索页面，这个页面允许用户搜索汽车同时可以根据颜色来过滤。颜色的选项是通过聚合获得的：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "query": {
    "match": {
      "make": "ford"
    }
  },
  "post_filter": {
    "term" : {
      "color" : "green"
    }
  },
  "aggs" : {
    "all_colors": {
      "terms" : { "field" : "color" }
    }
  }
}
```

post_filter 元素是 top-level 而且仅对命中结果进行过滤。

查询部分找到所有的 ford 汽车，然后用 terms 聚合创建一个颜色列表。因为聚合对查询范围进行操作，颜色列表与福特汽车有的颜色相对应。

最后，post_filter 会过滤搜索结果，只展示绿色 ford 汽车。这在查询执行过后发生，所以聚合不受影响。

这通常对 UI 的连贯一致性很重要，可以想象用户在界面商选择了一类颜色（比如：绿色），期望的是搜索结果已经被过滤了，而不是过滤界面上的选项。如果我们应用 filter 查询，界面会马上变成只显示绿色作为选项，这不是用户想要的！

性能考虑 (Performance consideration)

当你需要对搜索结果和聚合结果做不同的过滤时，你才应该使用 `post_filter`，有时用户会在普通搜索使用 `post_filter`。不要这么做！`post_filter` 的特性是在查询之后执行，任何过滤对性能带来的好处（比如缓存）都会完全失去。在我们需要不同过滤时，`post_filter` 只与聚合一起使用。

Copyright © Songbai Yang all right reserved, powered by Gitbook 该文件修订时间：2021-07-14

- [1. 小结](#)

1. 小结

选择合适类型的过滤（如：搜索命中、聚合或两者兼有）通常和我们期望如何表现用户交互有关。选择合适的过滤器（或组合）取决于我们期望如何将结果呈现给用户。

在 filter 过滤中的 non-scoring 查询，同时影响搜索结果和聚合结果。filter 桶影响聚合。post_filter 只影响搜索结果。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-14

- 1. 多桶排序

1. 多桶排序

多值桶（ terms 、 histogram 和 date_histogram ）动态生成很多桶。

Elasticsearch 是如何决定这些桶展示给用户的顺序呢？

默认的，桶会根据 doc_count 降序排列。这是一个好的默认行为，因为通常我们想要找到文档中与查询条件相关的最大值：售价、人口数量、频率。但有些时候我们希望能修改这个顺序，不同的桶有着不同的处理方式。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-14

- 1. 内置排序

1. 内置排序

这些排序模式是桶固有的能力：它们操作桶生成的数据，比如 doc_count。它们共享相同的语法，但是根据使用桶的不同会有些细微差别。

让我们做一个 terms 聚合但是按 doc_count 值的升序排序：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "colors" : {
      "terms" : {
        "field" : "color",
        "order": {
          "_count" : "asc"
        }
      }
    }
  }
}
```

我们为聚合引入了一个 order 对象，它允许我们可以根据以下几个值中的一个值进行排序：

_count

按文档数排序。对 terms、histogram、date_histogram 有效。

_term

按词项的字符串值的字母顺序排序。只在 terms 内使用。

_key

按每个桶的键值数值排序（理论上与 _term 类似）。只在 histogram 和 date_histogram 内使用。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-07-14

- 1. 按度量排序

1. 按度量排序

有时，我们会想基于度量计算的结果值进行排序。在我们的汽车销售分析仪表盘中，我们可能想按照汽车颜色创建一个销售条状图表，但按照汽车平均售价的升序进行排序。

我们可以增加一个度量，再指定 order 参数引用这个度量即可：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "colors" : {
      "terms" : {
        "field" : "color",
        "order": {
          "avg_price" : "asc"
        }
      },
      "aggs": {
        "avg_price": {
          "avg": {"field": "price"}
        }
      }
    }
  }
}
```

计算每个桶的平均售价。

桶按照计算平均值的升序排序。

我们可以采用这种方式用任何度量排序，只需简单的引用度量的名字。不过有些度量会输出多个值。extended_stats 度量是一个很好的例子：它输出好几个度量值。

如果我们想使用多值度量进行排序，我们只需以关心的度量为关键词使用点式路径：

```
GET /cars/transactions/_search
{
    "size" : 0,
    "aggs" : {
        "colors" : {
            "terms" : {
                "field" : "color",
                "order": {
                    "stats.variance" : "asc"
                }
            },
            "aggs": {
                "stats": {
                    "extended_stats": {"field": "price"}
                }
            }
        }
    }
}
```

使用 . 符号，根据感兴趣的度量进行排序。

在上面这个例子中，我们按每个桶的方差来排序，所以这种颜色售价方差最小的会排在结果集最前面。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-07-14

- 1. 基于“深度”度量排序

1. 基于“深度”度量排序

在前面的示例中，度量是桶的直接子节点。平均售价是根据每个 term 来计算的。

在一定条件下，我们也有可能对更深的度量进行排序，比如孙子桶或从孙桶。

我们可以定义更深的路径，将度量用尖括号 (>) 嵌套起来，像这样：

my_bucket>another_bucket>metric。

需要提醒的是嵌套路径上的每个桶都必须是单值的。filter 桶生成一个单值桶：所有与过滤条件匹配的文档都在桶中。多值桶（如：terms）动态生成许多桶，无法通过指定一个确定路径来识别。

目前，只有三个单值桶：filter、global 和 reverse_nested。让我们快速用示例说明，创建一个汽车售价的直方图，但是按照红色和绿色（不包括蓝色）车各自的方差来排序：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "colors" : {
      "histogram" : {
        "field" : "price",
        "interval": 2000,
        "order": {
          "red_green_cars>stats.variance" : "asc"
        }
      },
      "aggs": {
        "red_green_cars": {
          "filter": { "terms": {"color": ["red", "green"]}}, 
          "aggs": {
            "stats": {"extended_stats": {"field" : "price"}}
          }
        }
      }
    }
  }
}
```

按照嵌套度量的方差对桶的直方图进行排序。

因为我们使用单值过滤器 filter，我们可以使用嵌套排序。

按照生成的度量对统计结果进行排序。

本例中，可以看到我们如何访问一个嵌套的度量。stats 度量是 red_green_cars 聚合的子节点，而 red_green_cars 又是 colors 聚合的子节点。为了根据这个度量排序，我们定义了路径 red_green_cars>stats.variance。我们可以这么做，因为 filter 桶是个单值桶。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-07-14

- 1. 近似聚合

1. 近似聚合

如果所有的数据都在一台机器上，那么生活会容易许多。CS201课上教的经典算法就足够应付这些问题。如果所有的数据都在一台机器上，那么也就不需要像Elasticsearch这样的分布式软件了。不过一旦我们开始分布式存储数据，就需要小心地选择算法。

有些算法可以分布执行，到目前为止讨论过的所有聚合都是单次请求获得精确结果的。这些类型的算法通常被认为是高度并行的，因为它们无须任何额外代价，就能在多台机器上并行执行。比如当计算max度量时，以下的算法就非常简单：

把请求广播到所有分片。查看每个文档的price字段。如果 $price > current_max$ ，将 $current_max$ 替换成 $price$ 。返回所有分片的最大 price 并传给协调节点。找到从所有分片返回的最大 price。这是最终的最大值。这个算法可以随着机器数的线性增长而横向扩展，无须任何协调操作（机器之间不需要讨论中间结果），而且内存消耗很小（一个整型就能代表最大值）。

不幸的是，不是所有的算法都像获取最大值这样简单。更加复杂的操作则需要在算法的性能和内存使用上做出权衡。对于这个问题，我们有个三角因子模型：大数据、精确性和实时性。

我们需要选择其中两项：

精确 + 实时 数据可以存入单台机器的内存之中，我们可以随心所欲，使用任何想用的算法。结果会 100% 精确，响应会相对快速。大数据 + 精确 传统的 Hadoop。可以处理 PB 级的数据并且为我们提供精确的答案，但它可能需要几周的时间才能为我们提供这个答案。大数据 + 实时 近似算法为我们提供准确但不精确的结果。Elasticsearch 目前支持两种近似算法（cardinality 和 percentiles）。它们会提供准确但不是 100% 精确的结果。以牺牲一点小小的估算错误为代价，这些算法可以为我们换来高速的执行效率和极小的内存消耗。

对于大多数应用领域，能够实时返回高度准确的结果要比 100% 精确结果重要得多。乍一看这可能是天方夜谭。有人会叫“我们需要精确的答案！”但仔细考虑 0.5% 误差所带来的影响：

99% 的网站延时都在 132ms 以下。0.5% 的误差对以上延时的影响在正负 0.66ms。近似计算的结果会在毫秒内返回，而“完全正确”的结果就可能需要几秒，甚至无法返回。只要简单的查看网站的延时情况，难道我们会在意近似结果是 132.66ms 而不是 132ms 吗？当然，不是所有的领域都能容忍这种近似结果，但对于绝大多数来说是没有问题的。接受近似结果更多的是一种文化观念上的壁垒而不是商业或技术上的需要。

Copyright © Songbai Yang all right reserved, powered by Gitbook 该文件修订时间：2021-07-15

- 1. 统计去重后的数量
 - 1.1. 学会权衡
 - 1.2. 速度优化

1. 统计去重后的数量

Elasticsearch 提供的首个近似聚合是 cardinality (注：基数) 度量。它提供一个字段的基数，即该字段的 distinct 或者 unique 值的数目。你可能会对 SQL 形式比较熟悉：

```
SELECT COUNT(DISTINCT color)
FROM cars
```

去重是一个很常见的操作，可以回答很多基本的业务问题：

- 网站独立访客是多少？
- 卖了多少种汽车？
- 每月有多少独立用户购买了商品？

我们可以用 cardinality 度量确定经销商销售汽车颜色的数量：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "distinct_colors" : {
      "cardinality" : {
        "field" : "color"
      }
    }
  }
}
```

返回的结果表明已经售卖了三种不同颜色的汽车：

```
...
"aggregations": {
  "distinct_colors": {
    "value": 3
  }
}
...
```

可以让我们的例子变得更有用：每月有多少颜色的车被售出？为了得到这个度量，我们只需要将一个 cardinality 度量嵌入一个 date_histogram：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "months" : {
      "date_histogram": {
        "field": "sold",
        "interval": "month"
      },
      "aggs": {
        "distinct_colors" : {
          "cardinality" : {
            "field" : "color"
          }
        }
      }
    }
  }
}
```

1.1. 学会权衡

正如我们本章开头提到的，`cardinality` 度量是一个近似算法。它是基于 [HyperLogLog++](#) (HLL) 算法的。HLL 会先对我们的输入作哈希运算，然后根据哈希运算的结果中的 bits 做概率估算从而得到基数。

我们不需要理解技术细节（如果确实感兴趣，可以阅读这篇论文），但我们最好应该关注一下这个算法的 特性：

可配置的精度，用来控制内存的使用（更精确 = 更多内存）。小的数据集精度是非常高的。我们可以通过配置参数，来设置去重需要的固定内存使用量。无论数千还是数十亿的唯一值，内存使用量只与你配置的精确度相关。要配置精度，我们必须指定 `precision_threshold` 参数的值。这个阈值定义了在何种基数水平下我们希望得到一个近乎精确的结果。参考以下示例：

```
GET /cars/transactions/_search
{
  "size" : 0,
  "aggs" : {
    "distinct_colors" : {
      "cardinality" : {
        "field" : "color",
        "precision_threshold" : 100
      }
    }
  }
}
```

`precision_threshold` 接受 0–40,000 之间的数字，更大的值还是会当作 40,000 来处理。

示例会确保当字段唯一值在 100 以内时会得到非常准确的结果。尽管算法是无法保证这点的，但如果基数在阈值以下，几乎总是 100% 正确的。高于阈值的基数会开始节省内存而牺牲准确度，同时也会对度量结果带入误差。

对于指定的阈值，HLL 的数据结构会大概使用 `precision_threshold * 8` 字节的内存，所以就必须在牺牲内存和获得额外的准确度间做平衡。

在实际应用中，100 的阈值可以在唯一值为百万的情况下仍然将误差维持 5% 以内。

1.2. 速度优化

如果想要获得唯一值的数目，通常需要查询整个数据集合（或几乎所有数据）。所有基于所有数据的操作都必须迅速，原因是显然的。HyperLogLog 的速度已经很快了，它只是简单的对数据做哈希以及一些位操作。

但如果速度对我们至关重要，可以做进一步的优化。因为 HLL 只需要字段内容的哈希值，我们可以在索引时就预先计算好。就能在查询时跳过哈希计算然后将哈希值从 `fielddata` 直接加载出来。

注意

预先计算哈希值只对内容很长或者基数很高的字段有用，计算这些字段的哈希值的消耗在查询时是无法忽略的。尽管数值字段的哈希计算是非常快速的，存储它们的原始值通常需要同样（或更少）的内存空间。这对低基数的字符串字段同样适用，Elasticsearch 的内部优化能够保证每个唯一值只计算一次哈希。基本上说，预先计算并不能保证所有的字段都更快，它只对那些具有高基数和/或者内容很长的字符串字段有作用。需要记住的是，预算只是简单的将查询消耗的时间提前转移到索引时，并非没有任何代价，区别在于你可以选择在什么时候做这件事，要么在索引时，要么在查询时。

要想这么做，我们需要为数据增加一个新的多值字段。我们先删除索引，再增加一个包括哈希值字段的映射，然后重新索引：

```
DELETE /cars/  
  
PUT /cars/  
{  
  "mappings": {  
    "transactions": {  
      "properties": {  
        "color": {  
          "type": "string",  
          "fields": {  
            "hash": {  
              "type": "murmur3"  
            }  
          }  
        }  
      }  
    }  
  }  
}  
  
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" }
```

多值字段的类型是 murmur3 , 这是一个哈希函数。

现在当我们执行聚合时，我们使用 color.hash 字段而不是 color 字段：

```
GET /cars/transactions/_search  
{  
  "size" : 0,  
  "aggs" : {  
    "distinct_colors" : {  
      "cardinality" : {  
        "field" : "color.hash"  
      }  
    }  
  }  
}
```

注意我们指定的是哈希过的多值字段，而不是原始字段。

现在 cardinality 度量会读取 "color.hash" 里的值（预先计算的哈希值），取代动态计算原始值的哈希。

单个文档节省的时间是非常少的，但是如果你聚合一亿数据，每个字段多花费 10 纳秒的时间，那么在每次查询时都会额外增加 1 秒，如果我们要在非常大量的数据里面使用 cardinality，我们可以权衡使用预计算的意义，是否需要提前计算 hash，从而在查询时获得更好的性能，做一些性能测试来检验预计算哈希是否适用于你的应用场景。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-15

- 1. 百分位计算
 - 1.1. 百分位度量
 - 1.2. 百分位等级
 - 1.3. 学会权衡

1. 百分位计算

Elasticsearch 提供的另外一个近似度量就是 percentiles 百分位数度量。百分位数展现某以具体百分比下观察到的数值。例如，第95个百分位上的数值，是高于95% 的数据总和。

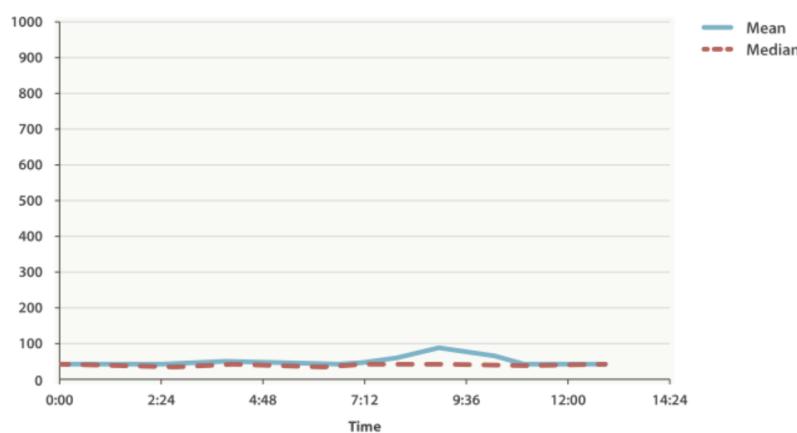
百分位数通常用来找出异常。在（统计学）的正态分布下，第 0.13 和 第 99.87 的百分位数代表与均值距离三倍标准差的值。任何处于三倍标准差之外的数据通常被认为是不寻常的，因为它与平均值相差太大。

更具体的说，假设我们正运行一个庞大的网站，一个很重要的工作是保证用户请求能得到快速响应，因此我们就需要监控网站的延时来判断响应是否能保证良好的用户体验。

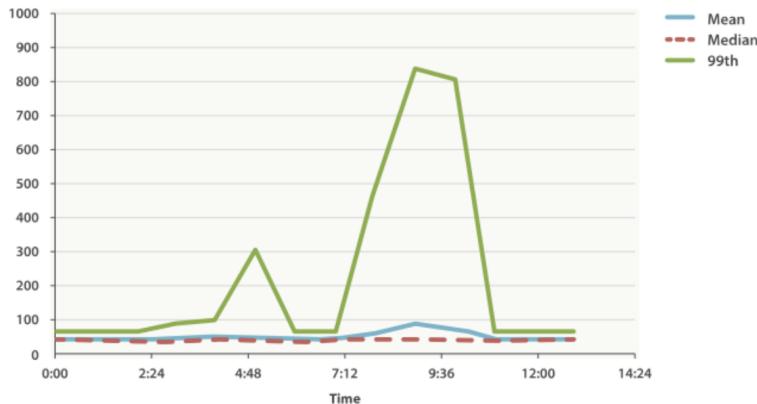
在此场景下，一个常用的度量方法就是平均响应延时。但这并不是一个好的选择（尽管很常用），因为平均数通常会隐藏那些异常值，中位数有着同样的问题。我们可以尝试最大值，但这个度量会轻而易举的被单个异常值破坏。

在图 Figure 40, “Average request latency over time” 查看问题。如果我们依靠如平均值或中位数这样的简单度量，就会得到像这样一幅图 Figure 40, “Average request latency over time”。

Figure 40. Average request latency over time



一切正常。图上有轻微的波动，但没有什么值得关注的。但如果我们将 99 百分位数时（这个值代表最慢的 1% 的延时），我们看到了完全不同的一幅画面，如图 Figure 41, “Average request latency with 99th percentile over time”。

Figure 41. Average request latency with 99th percentile over time

令人吃惊！在上午九点半时，均值只有 75ms。如果作为一个系统管理员，我们都不会看他第二眼。一切正常！但 99 百分位告诉我们有 1% 的用户碰到的延时超过 850ms，这是另外一幅场景。在上午4点48时也有一个小波动，这甚至无法从平均值和中位数曲线上观察到。

这只是百分位的一个应用场景，百分位还可以被用来快速用肉眼观察数据的分布，检查是否有数据倾斜或双峰甚至更多。

1.1. 百分位度量

让我加载一个新的数据集（汽车的数据不太适用于百分位）。我们要索引一系列网站延时数据然后运行一些百分位操作进行查看：

```
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" }
```

数据有三个值：延时、数据中心的区域以及时间戳。让我们对数据全集进行百分位操作以获得数据分布情况的直观感受：

```
GET /website/logs/_search
{
    "size" : 0,
    "aggs" : {
        "load_times" : {
            "percentiles" : {
                "field" : "latency"
            }
        },
        "avg_load_time" : {
            "avg" : {
                "field" : "latency"
            }
        }
    }
}
```

percentiles 度量被应用到 latency 延时字段。

为了比较，我们对相同字段使用 avg 度量。

默认情况下，percentiles 度量会返回一组预定义的百分位数值：[1, 5, 25, 50, 75, 95, 99]。它们表示了人们感兴趣的常用百分位数值，极端的百分位数在范围的两边，其他的一些处于中部。在返回的响应中，我们可以看到最小延时在 75ms 左右，而最大延时差不多有 600ms。与之形成对比的是，平均延时在 200ms 左右，信息并不是很多：

```
...
"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
    }
}
```

所以显然延时的分布很广，让我们看看它们是否与数据中心的地理区域有关：

```
GET /website/logs/_search
{
    "size" : 0,
    "aggs" : {
        "zones" : {
            "terms" : {
                "field" : "zone"
            },
            "aggs" : {
                "load_times" : {
                    "percentiles" : {
                        "field" : "latency",
                        "percents" : [50, 95.0, 99.0]
                    }
                },
                "load_avg" : {
                    "avg" : {
                        "field" : "latency"
                    }
                }
            }
        }
    }
}
```

首先根据区域我们将延时分到不同的桶中。

再计算每个区域的百分位数值。

percents 参数接受了我们想返回的一组百分位数，因为我们只对长的延时感兴趣。

在响应结果中，我们发现欧洲区域（EU）要比美国区域（US）慢很多，在美国区域（US），50 百分位与 99 百分位十分接近，它们都接近均值。

与之形成对比的是，欧洲区域（EU）在 50 和 99 百分位有较大区分。现在，显然可以发现是欧洲区域（EU）拉低了延时的统计信息，我们知道欧洲区域的 50% 延时都在 300ms+。

```

...
"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
        }
      }
    ]
  }
}
...

```

1.2. 百分位等级

这里有另外一个紧密相关的度量叫 percentile_ranks。percentiles 度量告诉我们落在某个百分比以下的所有文档的最小值。例如，如果 50 百分位是 119ms，那么有 50% 的文档数值都不超过 119ms。percentile_ranks 告诉我们某个具体值属于哪个百分位。119ms 的 percentile_ranks 是在 50 百分位。这基本是个双向关系，例如：

50 百分位是 119ms。119ms 百分位等级是 50 百分位。所以假设我们网站必须维持的服务等级协议 (SLA) 是响应时间低于 210ms。然后，开个玩笑，我们老板警告我们如果响应时间超过 800ms 会把我开除。可以理解的是，我们希望知道有多少百分比的请求可以满足 SLA 的要求（并期望至少在 800ms 以下！）。

为了做到这点，我们可以应用 percentile_ranks 度量而不是 percentiles 度量：

```
GET /website/logs/_search
{
    "size" : 0,
    "aggs" : {
        "zones" : {
            "terms" : {
                "field" : "zone"
            },
            "aggs" : {
                "load_times" : {
                    "percentile_ranks" : {
                        "field" : "latency",
                        "values" : [210, 800]
                    }
                }
            }
        }
    }
}
```

percentile_ranks 度量接受一组我们希望分级的数值。

在聚合运行后，我们能得到两个值：

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

这告诉我们三点重要的信息：

在欧洲 (EU) , 210ms 的百分位等级是 31.94% 。

在美国 (US) , 210ms 的百分位等级是 100% 。

在欧洲 (EU) 和美国 (US) , 800ms 的百分位等级是 100% 。

通俗的说，在欧洲区域 (EU) 只有 32% 的响应时间满足服务等级协议 (SLA) , 而美国区域 (US) 始终满足服务等级协议的。但幸运的是，两个区域所有响应时间都在 800ms 以下，所以我们还不会被炒鱿鱼（至少目前不会）。

percentile_ranks 度量提供了与 percentiles 相同的信息，但它以不同方式呈现，如果我们对某个具体数值更关心，使用它会更方便。

1.3. 学会权衡

和基数一样，计算百分位需要一个近似算法。朴素的实现会维护一个所有值的有序列表，但当我们有几十亿数据分布在几十个节点时，这几乎是不可能的。

取而代之的是 percentiles 使用一个 TDigest 算法，（由 Ted Dunning 在 Computing Extremely Accurate Quantiles Using T-Digests 里面提出的）。与 HyperLogLog 一样，不需要理解完整的技术细节，但有必要了解算法的特性：

- 百分位的准确度与百分位的极端程度相关，也就是说 1 或 99 的百分位要比 50 百分位要准确。这只是数据结构内部机制的一种特性，但这是一个好的特性，因为多数人只关心极端的百分位。
- 对于数值集合较小的情况，百分位非常准确。如果数据集足够小，百分位可能 100% 精确。
- 随着桶里数值的增长，算法会开始对百分位进行估算。它能有效在准确度和内存节省之间做出权衡。不准确的程度比较难以总结，因为它依赖于聚合时数据的分布以及数据量的大小。

与 cardinality 类似，我们可以通过修改参数 compression 来控制内存与准确度之间的比值。

TDigest 算法用节点近似计算百分比：节点越多，准确度越高（同时内存消耗也越大），这都与数据量成正比。compression 参数限制节点的最大数目为 $20 * compression$ 。

因此，通过增加压缩比值，可以以消耗更多内存为代价提高百分位数准确性。更大的压缩比值会使算法运行更慢，因为底层的树形数据结构的存储也会增长，也导致操作的代价更高。默认的压缩比值是 100。

一个节点大约使用 32 字节的内存，所以在最坏的情况下（例如，大量数据有序存入），默认设置会生成一个大小约为 64KB 的 TDigest。在实际应用中，数据会更随机，所以 TDigest 使用的内存会更少。

Copyright © Songbai Yang all right reserved, powered by Gitbook 该文件修订时间：2021-07-15

- 1. 通过聚合发现异常指标

1. 通过聚合发现异常指标

`significant_terms` (`SigTerms`) 聚合与其他聚合都不相同。目前为止我们看到的所有聚合在本质上都是简单的数学计算。将不同这些构造块相互组合在一起，我们可以创建复杂的聚合以及数据报表。

`significant_terms` 有着不同的工作机制。对有些人来说，它甚至看起来有点像机器学习。`significant_terms` 聚合可以在你数据集中找到一些 异常 的指标。

如何解释这些 不常见的 行为？这些异常的数据指标通常比我们预估出现的频次要更频繁，这些统计上的异常指标通常象征着数据里的某些有趣信息。

例如，假设我们负责检测和跟踪信用卡欺诈，客户打电话过来抱怨他们信用卡出现异常交易，它们的帐户已经被盗用。这些交易信息只是更严重问题的症状。在最近的某些地区，一些商家有意的盗取客户的信用卡信息，或者它们自己的信息无意中也被盗取。

我们的任务是找到 危害的共同点，如果我们有 100 个客户抱怨交易异常，他们很有可能都属于同一个商户，而这家商户有可能就是罪魁祸首。

当然，这里面还有一些特例。例如，很多客户在它们近期交易历史记录中会有很大的商户如亚马逊，我们可以将亚马逊排除在外，然而，在最近一些有问题的信用卡的商家里面也有亚马逊。

这是一个 普通的共同 商户的例子。每个人都共享这个商户，无论有没有遭受危害。我们对它并不感兴趣。

相反，我们有一些很小商户比如街角的一家药店，它们属于 普通但不寻常 的情况，只有一两个客户有交易记录。我们同样可以将这些商户排除，因为所有受到危害的信用卡都没有与这些商户发生过交易，我们可以肯定它们不是安全漏洞的责任方。

我们真正想要的是 不普通的共同 商户。所有受到危害的信用卡都与它们发生过交易，但是在未受危害的背景噪声下，它们并不明显。这些商户属于统计异常，它们比应该出现的频率要高。这些不普通的共同商户很有可能就是需要调查的。

`significant_terms` 聚合就是做这些事情。它分析统计你的数据并通过对比正常数据找到可能有异常频次的指标。

暴露的异常指标 代表什么 依赖的你数据。对于信用卡数据，我们可能会想找出信用卡欺诈。对于电商数据，我们可能会想找出未被识别的人口信息，从而进行更高效的市场推广。如果我们正在分析日志，我们可能会发现一个服务器会抛出比它本应抛出的更多异常。`significant_terms` 的应用还远不止这些。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-07-15

- [1. significant_terms 演示](#)
 - [1.1. 基于流行程度推荐](#)
 - [1.2. 基于统计的推荐](#)

1. significant_terms 演示

因为 significant_terms 聚合是通过分析统计信息来工作的，需要为数据设置一个阀值让它们更有效。也就是说无法通过只索引少量示例数据来展示它。

正因如此，我们准备了一个大约 8000 个文档的数据集，并将它的快照保存在一个公共演示库中。可以通过以下步骤在集群中还原这些数据：

在 `elasticsearch.yml` 配置文件中增加以下配置，以便将演示库加入到白名单中：

```
repositories.url.allowed_urls: ["http://download.elastic.co/*"]
```

重启 Elasticsearch。运行以下快照命令。（更多使用快照的信息，参见 [备份集群（Backing Up Your Cluster）](#)）。

```
PUT /_snapshot/sigterms
{
  "type": "url",
  "settings": {
    "url": "http://download.elastic.co/definitiveguide/sigterms_demo/"
  }
}

GET /_snapshot/sigterms/_all

POST /_snapshot/sigterms/snapshot/_restore

GET /mlmovies,mlratings/_recovery
```

注册一个新的只读地址库，并指向演示快照。

(可选) 检查库内关于快照的详细信息。

开始还原过程。会在集群中创建两个索引：`mlmovies` 和 `mlratings`。

(可选) 使用 Recovery API 监控还原过程。

注意

数据集有 50 MB 会需要一些时间下载。

在本演示中，会看看 MovieLens 里面用户对电影的评分。在 MovieLens 里，用户可以推荐电影并评分，这样其他用户也可以找到新的电影。为了演示，会基于输入的电影采用 `significant_terms` 对电影进行推荐。

让我们看看示例中的数据，感受一下要处理的内容。本数据集有两个索引，`mlmovies` 和 `mlratings`。首先查看 `mlmovies`：

```
GET mlmovies/_search

{
  "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)"
        }
      },
      ...
    ]
  }
}
```

执行一个不带查询条件的搜索，以便能看到一组随机演示文档。

mlmovies 里的每个文档表示一个电影，数据有两个重要字段：电影ID _id 和电影名 title。可以忽略 offset 和 bytes。它们是从原始 CSV 文件抽取数据的过程中产生的中间属性。数据集中有 10,681 部影片。

现在来看看 mlratings：

```
GET mlratings/_search

{
  "took": 3,
  "timed_out": false,
  "_shards": {...},
  "hits": {
    "total": 69796,
    "max_score": 1,
    "hits": [
      {
        "_index": "mlratings",
        "_type": "mlrating",
        "_id": "00IC-2jDQFkQkpD6vhbFYA",
        "_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
        }
      },
      ...
    ]
  }
}
```

这里可以看到每个用户的推荐信息。每个文档表示一个用户，用 ID 字段 user 来表示， movie 字段维护一个用户观看和推荐过的影片列表。

1.1. 基于流行程度推荐

可以采取的首个策略就是基于流行程度向用户推荐影片。对于某部影片，找到所有推荐过它的用户，然后将他们的推荐进行聚合并获得推荐中最流行的五部。

我们可以很容易的通过一个 terms 聚合 以及一些过滤来表示它，看看 Talladega Nights（塔拉迪加之夜）这部影片，它是 Will Ferrell 主演的一部关于全国运动汽车竞赛（NASCAR racing）的喜剧。在理想情况下，我们的推荐应该找到类似风格的喜剧（很有可能也是 Will Ferrell 主演的）。

首先需要找到影片 Talladega Nights 的 ID：

```
GET mlmovies/_search
{
  "query": {
    "match": {
      "title": "Talladega Nights"
    }
  }
}

...
"hits": [
  {
    "_index": "mlmovies",
    "_type": "mlmovie",
    "_id": "46970",
    "_score": 3.658795,
    "_source": {
      "offset": 9575,
      "bytes": 74,
      "title": "Talladega Nights: The Ballad of Ricky Bobby (2006)"
    }
  },
  ...
]
```

Talladega Nights 的 ID 是 46970。

有了 ID，可以过滤评分，再应用 terms 聚合从喜欢 Talladega Nights 的用户中找到最流行的影片：

```
GET mlratings/_search
{
  "size" : 0,
  "query": {
    "filtered": {
      "filter": {
        "term": {
          "movie": 46970
        }
      }
    }
  },
  "aggs": {
    "most_popular": {
      "terms": {
        "field": "movie",
        "size": 6
      }
    }
  }
}
```

这次查询 mlratings， 将结果内容 大小设置 为 0 因为我们只对聚合的结果感兴趣。对影片 Talladega Nights 的 ID 使用过滤器。最后，使用 terms 桶找到最流行的影片。

在 mlratings 索引下搜索，然后对影片 Talladega Nights 的 ID 使用过滤器。由于聚合是针对查询范围进行操作的，它可以有效的过滤聚合结果从而得到那些只推荐 Talladega Nights 的用户。最后，执行 terms 聚合得到最流行的影片。请求排名最前的六个结果，因为 Talladega Nights 本身很有可能就是其中一个结果（并不想重复推荐它）。

返回结果就像这样：

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

通过一个简单的过滤查询，将得到的结果转换成原始影片名：

```
GET mlmovies/_search  
{  
    "query": {  
        "filtered": {  
            "filter": {  
                "ids": {  
                    "values": [2571,318,296,2959,260]  
                }  
            }  
        }  
    }  
}
```

最后得到以下列表：

- Matrix, The (黑客帝国)
- Shawshank Redemption (肖申克的救赎)
- Pulp Fiction (低俗小说)
- Fight Club (搏击俱乐部)

- Star Wars Episode IV: A New Hope (星球大战 IV: 曙光乍现)

好吧，这肯定不是一个好的列表！我喜欢所有这些影片。但问题是：几乎每个人都喜欢它们。这些影片本来就受大众欢迎，也就是说它们出现在每个人的推荐中都会受欢迎。这其实是一个流行影片的推荐列表，而不是和影片 Talladega Nights 相关的推荐。

可以通过再次运行聚合轻松验证，而不需要对影片 Talladega Nights 进行过滤。会提供最流行影片的前五名列表：

```
GET mlratings/_search
{
  "size" : 0,
  "aggs": {
    "most_popular": {
      "terms": {
        "field": "movie",
        "size": 5
      }
    }
  }
}
```

返回列表非常相似：

- Shawshank Redemption (肖申克的救赎)
- Silence of the Lambs, The (沉默的羔羊)
- Pulp Fiction (低俗小说)
- Forrest Gump (阿甘正传)
- Star Wars Episode IV: A New Hope (星球大战 IV: 曙光乍现)

显然，只是检查最流行的影片是不能足以创建一个良好而又具鉴别能力的推荐系统。

1.2. 基于统计的推荐

现在场景已经设定好，使用 significant_terms。significant_terms 会分析喜欢影片 Talladega Nights 的用户组（前端 用户组），并且确定最流行的电影。然后为每个用户（后端 用户）构造一个流行影片列表，最后将两者进行比较。

统计异常就是与统计背景相比在前景特征组中过度展现的那些影片。理论上讲，它应该是一组喜剧，因为喜欢 Will Ferrell 喜剧的人给这些影片的评分会比一般人高。

让我们试一下：

```
GET mlratings/_search
{
  "size" : 0,
  "query": {
    "filtered": {
      "filter": {
        "term": {
          "movie": 46970
        }
      }
    }
  },
  "aggs": {
    "most_sig": {
      "significant_terms": {
        "field": "movie",
        "size": 6
      }
    }
  }
}
```

设置几乎一模一样，只是用 significant_terms 替代了 terms。

正如所见，查询也几乎是一样的。过滤出喜欢影片 Talladega Nights 的用户，他们组成了前景特征用户组。默认情况下，significant_terms 会使用整个索引里的数据作为统计背景，所以不需要特别的处理。

与 terms 类似，结果返回了一组桶，不过有更多的元数据信息：

```
...
  "aggregations": {
    "most_sig": {
      "doc_count": 271,
      "buckets": [
        {
          "key": 46970,
          "key_as_string": "46970",
          "doc_count": 271,
          "score": 256.549815498155,
          "bg_count": 271
        },
        {
          "key": 52245,
          "key_as_string": "52245",
          "doc_count": 59,
          "score": 17.66462367106966,
          "bg_count": 185
        },
        {
          "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
        }
      ]
    ...
  }
```

顶层 doc_count 展现了前景特征组里文档的数量。

每个桶里面列出了聚合的键值（例如，影片的ID）。

桶内文档的数量 doc_count。

背景文档的数量，表示该值在整个统计背景里出现的频度。

可以看到，获得的第一个桶是 Talladega Nights。它可以在所有 271 个文档中找到，这并不意外。让我们看下一个桶：键值 52245。

这个 ID 对应影片 Blades of Glory（荣誉之刃），它是一部关于男子学习滑冰的喜剧，也是由 Will Ferrell 主演。可以看到喜欢 Talladega Nights 的用户对它的推荐是 59 次。这也意味着 21% 的前景特征用户组推荐了影片 Blades of Glory（59 /

$271 = 0.2177$) 。

形成对比的是， Blades of Glory 在整个数据集合中仅被推荐了 185 次， 只占 0.26% ($185 / 69796 = 0.00265$) 。因此 Blades of Glory 是一个统计异常：它在喜欢 Talladega Nights 的用户中是显著的共性（注：uncommonly common）。这样就找到了一个好的推荐！

如果看完整的列表，它们都是好的喜剧推荐（其中很多也是由 Will Ferrell 主演）：

Blades of Glory (荣誉之刃) Anchorman: The Legend of Ron Burgundy (王牌播音员) Semi-Pro (半职业选手) Knocked Up (一夜大肚) 40-Year-Old Virgin, The (四十岁的老处男) 这只是 significant_terms 它强大的一个示例，一旦开始使用 significant_terms，可能碰到这样的情况，我们不想要最流行的，而想要显著的共性（注：uncommonly common）。这个简单的聚合可以揭示出一些数据里出人意料的复杂趋势。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-07-16

- [1. Doc Values and Fielddata](#)

1. Doc Values and Fielddata

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-07-22

- [1. Doc Values](#)

1. Doc Values

聚合使用一个叫 doc values 的数据结构（在 [Doc Values 介绍](#) 里简单介绍）。 Doc values 可以使聚合更快、更高效并且内存友好，所以理解它的工作方式十分有益。

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

对于以下倒排索引：

```
| Term | Doc_1 | Doc_2 | Doc_3 | | ---- | ---- | ---- | | brown | X | X | | | dog | X | |
x | | dogs | | X | x | | fox | X | | x | | foxes | | X | | | in | | X | | | jumped | X | | x | | lazy |
X | X | | | leap | | X | | | over | X | X | x | | quick | X | X | x | | summer | | X | | | the | X
| | x |
```

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

```
GET /my_index/_search
{
  "query" : {
    "match" : {
      "body" : "brown"
    }
  },
  "aggs" : {
    "popular_terms": {
      "terms" : {
        "field" : "body"
      }
    }
  }
}
```

查询部分简单又高效。倒排索引是根据项来排序的，所以我们首先在词项列表中找到 brown，然后扫描所有列，找到包含 brown 的文档。我们可以快速看到 Doc_1 和 Doc_2 包含 brown 这个 token。

然后，对于聚合部分，我们需要找到 Doc_1 和 Doc_2 里所有唯一的词项。用倒排索引做这件事情代价很高：我们会迭代索引里的每个词项并收集 Doc_1 和 Doc_2 列里面 token。这很慢而且难以扩展：随着词项和文档的数量增加，执行时间也会增加。

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

```
| Doc | Terms | | ---- | ---- |
| Doc_1 | brown, dog, fox, jumped, lazy, over, quick, the | | Doc_2 | brown, dogs,
foxes, in, lazy, leap, over, quick, summer | | Doc_3 | dog, dogs, fox, jumped, over,
quick, the |
```

当数据被转置之后，想要收集到 Doc_1 和 Doc_2 的唯一 token 会非常容易。获得每个文档行，获取所有的词项，然后求两个集合的并集。

因此，搜索和聚合是相互紧密缠绕的。搜索使用倒排索引查找文档，聚合操作收集和聚合 doc values 里的数据。

注意

Doc values 不仅可以用于聚合。任何需要查找某个文档包含的值的操作都必须使用它。除了聚合，还包括排序，访问字段值的脚本，父子关系处理（参见 [父-子关系文档](#)）。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-07-16

- [1. 深入理解 Doc Values](#)
 - [1.1. 列式存储的压缩](#)
 - [1.2. 禁用Doc Values](#)

1. 深入理解 Doc Values

在上一节一开头我们就说 Doc Values 是 "快速、高效并且内存友好"。这个口号听起来不错，不过话说回来 Doc Values 到底是如何工作的呢？

Doc Values 是在索引时与 倒排索引 同时生成。也就是说 Doc Values 和 倒排索引一样，基于 Segement 生成并且是不可变的。同时 Doc Values 和 倒排索引一样序列化到磁盘，这样对性能和扩展性有很大帮助。

Doc Values 通过序列化把数据结构持久化到磁盘，我们可以充分利用操作系统的内存，而不是 JVM 的 Heap。当 working set 远小于系统的可用内存，系统会自动将 Doc Values 驻留在内存中，使得其读写十分快速；不过，当其远大于可用内存时，系统会根据需要从磁盘读取 Doc Values，然后选择性放到分页缓存中。很显然，这样性能会比在内存中差很多，但是它的大小就不再局限于服务器的内存了。如果是使用 JVM 的 Heap 来实现那么只能是因为 OutOfMemory 导致程序崩溃了。

注意

因为 Doc Values 不是由 JVM 来管理，所以 Elasticsearch 实例可以配置一个很小的 JVM Heap，这样给系统留出来更多的内存。同时更小的 Heap 可以让 JVM 更加快速和高效的回收。之前，我们会建议分配机器内存的 50% 来给 JVM Heap。但是对于 Doc Values，这样可能不是最合适的方案了。以 64gb 内存的机器为例，可能给 Heap 分配 4-16gb 的内存更合适，而不是 32gb。

有关更详细的讨论，查看 [堆内存:大小和交换](#)。

1.1. 列式存储的压缩

从广义来说，Doc Values 本质上是一个序列化的 列式存储。正如我们上一节所讨论的，列式存储适用于聚合、排序、脚本等操作。

而且，这种存储方式也非常便于压缩，特别是数字类型。这样可以减少磁盘空间并且提高访问速度。现代 CPU 的处理速度要比磁盘快几个数量级（尽管即将到来的 NVMe 驱动器正在迅速缩小差距）。所以我们必须减少直接存磁盘读取数据的大小，尽管需要额外消耗 CPU 运算来进行解压。

要了解它如何压缩数据的，来看一组数字类型的 Doc Values：

```
| Doc | Terms | | ---- | ---- |  
| Doc_1 | 100 || Doc_2 | 1000 || Doc_3 | 1500 || Doc_1 | 1200 || Doc_2 | 300 ||  
Doc_3 | 1900 |
```

按列布局意味着我们有一个连续的数据块：[100,1000,1500,1200,300,1900,4200]。因为我们已经知道他们都是数字（而不是像文档或行中看到的异构集合），所以我们可以使用统一的偏移来将他们紧紧排列。

而且，针对这样的数字有很多种压缩技巧。你会注意到这里每个数字都是 100 的倍数，Doc Values 会检测一个段里面的所有数值，并使用一个最大公约数，方便做进一步的数据压缩。

如果我们保存 100 作为此段的除数，我们可以对每个数字都除以 100，然后得到：[1, 10, 15, 12, 3, 19, 42]。现在这些数字变小了，只需要很少的位就可以存储下，也减少了磁盘存放的大小。

Doc Values 在压缩过程中使用如下技巧。它会按依次检测以下压缩模式：

- 如果所有的数值各不相同（或缺失），设置一个标记并记录这些值
- 如果这些值小于 256，将使用一个简单的编码表
- 如果这些值大于 256，检测是否存在一个最大公约数
- 如果没有存在最大公约数，从最小的数值开始，统一计算偏移量进行编码

你会发现这些压缩模式不是传统的通用的压缩方式，比如 DEFLATE 或是 LZ4。因为列式存储的结构是严格且良好定义的，我们可以通过使用专门的模式来达到比通用压缩算法（如 LZ4）更高的压缩效果。

注意

你也许会想“好吧，貌似对数字很好，不知道字符串怎么样？”通过借助顺序表（ordinal table），String 类型也是类似进行编码的。String 类型是去重之后存放到顺序表的，通过分配一个 ID，然后通过数字类型的 ID 构建 Doc Values。这样 String 类型和数值类型可以达到同样的压缩效果。

顺序表本身也有很多压缩技巧，比如固定长度、变长或是前缀字符编码等等。

1.2. 禁用Doc Values

Doc Values 默认对所有字段启用，除了 analyzed strings。也就是说所有的数字、地理坐标、日期、IP 和不分析（not_analyzed）字符串类型都会默认开启。

analyzed strings 暂时还不能使用 Doc Values。文本经过分析流程生成很多 Token，使得 Doc Values 不能高效运行。我们将在 聚合与分析 讨论如何使用分析字符串类型来做聚合。

因为 Doc Values 默认启用，你可以选择对你数据集里面的大多数字段进行聚合和排序操作。但是如果你知道你永远也不会对某些字段进行聚合、排序或是使用脚本操作？尽管这并不常见，但是你可以通过禁用特定字段的 Doc Values。这样不仅节省磁盘空间，也许会提升索引的速度。

要禁用 Doc Values，在字段的映射（mapping）设置 doc_values: false 即可。例如，这里我们创建了一个新的索引，字段 "session_id" 禁用了 Doc Values：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "session_id": {
          "type": "string",
          "index": "not_analyzed",
          "doc_values": false
        }
      }
    }
  }
}
```

通过设置 `doc_values: false`，这个字段将不能被用于聚合、排序以及脚本操作

反过来也是可以进行配置的：让一个字段可以被聚合，通过禁用倒排索引，使它不能被正常搜索，例如：

```
PUT my_index
{
  "mappings": {
    "my_type": {
      "properties": {
        "customer_token": {
          "type": "string",
          "index": "not_analyzed",
          "doc_values": true,
          "index": "no"
        }
      }
    }
  }
}
```

Doc Values 被启用来允许聚合

索引被禁用了，这让该字段不能被查询/搜索

通过设置 `doc_values: true` 和 `index: no`，我们得到一个只能被用于聚合/排序/脚本的字段。无可否认，这是一个非常少见的情况，但有时很有用。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-16

- 1. 聚合与分析
 - 1.1. 分析字符串和 Fielddata (Analyzed strings and Fielddata)
 - 1.2. 高基数内存的影响 (High-Cardinality Memory Implications)

1. 聚合与分析

有些聚合，比如 terms 桶，操作字符串字段。字符串字段可能是 analyzed 或者 not_analyzed，那么问题来了，分析是怎么影响聚合的呢？

答案是影响“很多”，有两个原因：分析影响聚合中使用的 tokens，并且 doc values 不能使用于分析字符串。

让我们解决第一个问题：分析 tokens 的产生如何影响聚合。首先索引一些代表美国各个州的文档：

```
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" }
```

我们希望创建一个数据集里各个州的唯一列表，并且计数。简单，让我们使用 terms 桶：

```
GET /agg_analysis/data/_search
{
  "size" : 0,
  "aggs" : {
    "states" : {
      "terms" : {
        "field" : "state"
      }
    }
  }
}
```

得到结果：

```
{  
...  
    "aggregations": {  
        "states": {  
            "buckets": [  
                {  
                    "key": "new",  
                    "doc_count": 5  
                },  
                {  
                    "key": "york",  
                    "doc_count": 3  
                },  
                {  
                    "key": "jersey",  
                    "doc_count": 1  
                },  
                {  
                    "key": "mexico",  
                    "doc_count": 1  
                }  
            ]  
        }  
    }  
}
```

宝贝儿，这完全不是我们想要的！没有对州名计数，聚合计算了每个词的数目。背后的原因很简单：聚合是基于倒排索引创建的，倒排索引是后置分析（post-analysis）的。

当我们把这些文档加入到 Elasticsearch 中时，字符串 "New York" 被分析/分析成 ["new", "york"]。这些单独的 tokens，都被用来填充聚合计数，所以我们最终看到 new 的数量而不是 New York。

这显然不是我们想要的行为，但幸运的是很容易修正它。

我们需要为 state 定义 multfield 并且设置成 not_analyzed。这样可以防止 New York 被分析，也意味着在聚合过程中它会以单个 token 的形式存在。让我们尝试完整的过程，但这次指定一个 raw multfield：

```
DELETE /agg_analysis/
PUT /agg_analysis
{
  "mappings": {
    "data": {
      "properties": {
        "state" : {
          "type": "string",
          "fields": {
            "raw" : {
              "type": "string",
              "index": "not_analyzed"
            }
          }
        }
      }
    }
  }
}

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" }

GET /agg_analysis/data/_search
{
  "size" : 0,
  "aggs" : {
    "states" : {
      "terms" : {
        "field" : "state.raw"
      }
    }
  }
}
```

这次我们显式映射 state 字段并包括一个 not_analyzed 辅字段。
聚合针对 state.raw 字段而不是 state 。

现在运行聚合， 我们得到了合理的结果：

```
{
...
"aggregations": {
  "states": {
    "buckets": [
      {
        "key": "New York",
        "doc_count": 3
      },
      {
        "key": "New Jersey",
        "doc_count": 1
      },
      {
        "key": "New Mexico",
        "doc_count": 1
      }
    ]
  }
}
}
```

在实际中，这样的问题很容易被察觉，我们的聚合会返回一些奇怪的桶，我们会记住分析的问题。总之，很少有在聚合中使用分析字段的实例。当我们疑惑时，只要增加一个 `multifield` 就能有两种选择。

1.1. 分析字符串和 Fielddata (Analyzed strings and Fielddata)

当第一个问题涉及如何聚合数据并显示给用户，第二个问题主要是技术和幕后。

`Doc values` 不支持 `analyzed` 字符串字段，因为它们不能很有效的表示多值字符串。`Doc values` 最有效的是，当每个文档都有一个或几个 `tokens` 时，但不是无数的，分析字符串（想象一个 PDF，可能有几兆字节并有数以千计的独特 `tokens`）。

出于这个原因，`doc values` 不生成分析的字符串，然而，这些字段仍然可以使用聚合，那怎么可能呢？

答案是一种被称为 `fielddata` 的数据结构。与 `doc values` 不同，`fielddata` 构建和管理 100% 在内存中，常驻于 JVM 内存堆。这意味着它本质上是不可扩展的，有很多边缘情况下要提防。本章的其余部分是解决在分析字符串上下文中 `fielddata` 的挑战。

note

从历史上看，`fielddata` 是所有字段的默认设置。但是 Elasticsearch 已迁移到 `doc values` 以减少 OOM 的几率。分析的字符串是仍然使用 `fielddata` 的最后一块阵地。最终目标是建立一个序列化的数据结构类似于 `doc values`，可以处理高维度的分析字符串，逐步淘汰 `fielddata`。

1.2. 高基数内存的影响 (High-Cardinality Memory Implications)

避免分析字段的另外一个原因就是：高基数字段在加载到 fielddata 时会消耗大量内存。分析的过程会经常（尽管不总是这样）生成大量的 token，这些 token 大多都是唯一的。这会增加字段的整体基数并且带来更大的内存压力。

有些类型的分析对于内存来说极度不友好，想想 n-gram 的分析过程，New York 会被 n-gram 分析成以下 token：

- ne
- ew
- w
- y
- yo
- or
- rk

可以想象 n-gram 的过程是如何生成大量唯一 token 的，特别是在分析成段文本的时候。当这些数据加载到内存中，会轻而易举的将我们堆空间消耗殆尽。

因此，在聚合字符串字段之前，请评估情况：

这是一个 not_analyzed 字段吗？如果是，可以通过 doc values 节省内存。
否则，这是一个 analyzed 字段，它将使用 fielddata 并加载到内存中。这个字段因为 ngrams 有一个非常大的基数？如果是，这对于内存来说极度不友好。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-07-16

- [1. 限制内存使用](#)
 - [1.1. Fielddata 的大小](#)
 - [1.2. 监控 fielddata](#)
- [2. 断路器](#)

1. 限制内存使用

一旦分析字符串被加载到 fielddata，他们会一直在那里，直到被驱逐（或者节点崩溃）。由于这个原因，留意内存的使用情况，了解它是如何以及何时加载的，怎样限制对集群的影响是很重要的。

Fielddata 是 延迟 加载。如果你从来没有聚合一个分析字符串，就不会加载 fielddata 到内存中。此外，fielddata 是基于字段加载的，这意味着只有很活跃地使用字段才会增加 fielddata 的负担。

然而，这里有一个令人惊讶的地方。假设你的查询是高度选择性和只返回命中的 100 个结果。大多数人认为 fielddata 只加载 100 个文档。

实际情况是，fielddata 会加载索引中（针对该特定字段的）所有的 文档，而不管查询的特异性。逻辑是这样：如果查询会访问文档 X、Y 和 Z，那很有可能会在下一个查询中访问其他文档。

与 doc values 不同，fielddata 结构不会在索引时创建。相反，它是在查询运行时，动态填充。这可能是一个比较复杂的操作，可能需要一些时间。将所有的信息一次加载，再将其维持在内存中的方式要比反复只加载一个 fielddata 的部分代价要低。

JVM 堆 是有限资源的，应该被合理利用。限制 fielddata 对堆使用的影响有多套机制，这些限制方式非常重要，因为堆栈的乱用会导致节点不稳定（感谢缓慢的垃圾回收机制），甚至导致节点宕机（通常伴随 OutOfMemory 异常）。

选择堆大小 (Choosing a Heap Size)

在设置 Elasticsearch 堆大小时需要通过 \$ES_HEAP_SIZE 环境变量应用两个规则：

不要超过可用 RAM 的 50%

Lucene 能很好利用文件系统的缓存，它是通过系统内核管理的。如果没有足够的文件系统缓存空间，性能会受到影响。此外，专用于堆的内存越多意味着其他所有使用 doc values 的字段内存越少。

不要超过 32 GB

如果堆大小小于 32 GB，JVM 可以利用指针压缩，这可以大大降低内存的使用：每个指针 4 字节而不是 8 字节。

更详细和更完整的堆大小讨论，请参阅 堆内存:大小和交换

1.1. Fielddata 的大小

`indices.fielddata.cache.size` 控制为 fielddata 分配的堆空间大小。当你发起一个查询，分析字符串的聚合将会被加载到 fielddata，如果这些字符串之前没有被加载过。如果结果中 fielddata 大小超过了指定大小，其他的值将会被回收从而获得空间。

默认情况下，设置都是 `unbounded`， Elasticsearch 永远都不会从 `fielddata` 中回收数据。

这个默认设置是刻意选择的： `fielddata` 不是临时缓存。它是驻留内存里的数据结构，必须可以快速执行访问，而且构建它的代价十分高昂。如果每个请求都重载数据，性能会十分糟糕。

一个有界的大小会强制数据结构回收数据。我们会看何时应该设置这个值，但请首先阅读以下警告：

注意

这个设置是一个安全卫士，而非内存不足的解决方案。

如果没有足够空间可以将 `fielddata` 保留在内存中，Elasticsearch 就会时刻从磁盘重载数据，并回收其他数据以获得更多空间。内存的回收机制会导致重度磁盘I/O，并且在内存中生成很多垃圾，这些垃圾必须在晚些时候被回收掉。

设想我们正在对日志进行索引，每天使用一个新的索引。通常我们只对过去一两天的数据感兴趣，尽管我们会保留老的索引，但我们很少需要查询它们。不过如果采用默认设置，旧索引的 `fielddata` 永远不会从缓存中回收！ `fielddata` 会保持增长直到 `fielddata` 发生断熔（请参阅 断路器），这样我们就无法载入更多的 `fielddata`。

这个时候，我们被困在了死胡同。但我们仍然可以访问旧索引中的 `fielddata`，也无法加载任何新的值。相反，我们应该回收旧的数据，并为新值获得更多空间。

为了防止发生这样的事情，可以通过在 `config/elasticsearch.yml` 文件中增加配置为 `fielddata` 设置一个上限：

```
indices.fielddata.cache.size: 20%
```

可以设置堆大小的百分比，也可以是某个值，例如：`5gb`。

有了这个设置，最久未使用（LRU）的 `fielddata` 会被回收为新数据腾出空间。



可能发现在线文档有另外一个设置：`indices.fielddata.cache.expire`。这个设置永远都不会被使用！它很有可能在不久的将来被弃用。这个设置要求 Elasticsearch 回收那些过期的 `fielddata`，不管这些值有没有被用到。

这对性能是一件很糟糕的事情。回收会有消耗性能，它刻意的安排回收方式，而没能获得任何回报。没有理由使用这个设置：我们不能从理论上假设一个有用的情形。目前，它的存在只是为了向前兼容。我们只在很有以前提到过这个设置，但不幸的是网上各种文章都将其作为一种性能调优的小窍门来推荐。它不是。永远不要使用！

1.2. 监控 fielddata

无论是仔细监控 `fielddata` 的内存使用情况，还是看有无数据被回收都十分重要。高的回收数可以预示严重的资源问题以及性能不佳的原因。

`Fielddata` 的使用可以被监控：

- 按索引使用 indices-stats API :

```
GET /_stats/fielddata?fields=*
```

- 按节点使用 nodes-stats API :

```
GET /_nodes/stats/indices/fielddata?fields=*
```

- 按索引节点

```
GET /_nodes/stats/indices/fielddata?level=indices&fields=*
```

使用设置 ?fields=* , 可以将内存使用分配到每个字段。

2. 断路器

机敏的读者可能已经发现 fielddata 大小设置的一个问题。fielddata 大小是在数据加载 之后 检查的。如果一个查询试图加载比可用内存更多的信息到 fielddata 中会发生什么？答案很丑陋：我们会碰到 OutOfMemoryException 。

Elasticsearch 包括一个 fielddata 断熔器，这个设计就是为了处理上述情况。断熔器通过内部检查（字段的类型、基数、大小等等）来估算一个查询需要的内存。它然后检查要求加载的 fielddata 是否会导致 fielddata 的总量超过堆的配置比例。

如果估算查询的大小超出限制，就会 触发 断路器，查询会被中止并返回异常。这都发生在数据加载 之前，也就意味着不会引起 OutOfMemoryException 。

可用的断路器 (Available Circuit Breakers)

Elasticsearch 有一系列的断路器，它们都能保证内存不会超出限制：
indices.breaker.fielddata.limit
fielddata 断路器默认设置堆的 60% 作为 fielddata 大小的上限。
indices.breaker.request.limit request 断路器估算需要完成其他请求部分的结构大小，例如创建一个聚合桶，默认限制是堆内存的 40%。
indices.breaker.total.limit total 揉合 request 和 fielddata 断路器保证两者组合起来不会使用超过堆内存的 70%。

断路器的限制可以在文件 config/elasticsearch.yml 中指定，可以动态更新一个正在运行的集群：

```
PUT /_cluster/settings
{
  "persistent" : {
    "indices.breaker.fielddata.limit" : "40%"
  }
}
```

这个限制是按对内存的百分比设置的。

最好为断路器设置一个相对保守点的值。记住 fielddata 需要与 request 断路器共享堆内存、索引缓冲内存和过滤器缓存。Lucene 的数据被用来构造索引，以及各种其他临时的数据结构。正因如此，它默认值非常保守，只有 60% 。过于乐观的设置可能会引起潜在的堆栈溢出 (OOM) 异常，这会使整个节点宕掉。

另一方面，过度保守的值只会返回查询异常，应用程序可以对异常做相应处理。异常比服务器崩溃要好。这些异常应该也能促进我们对查询进行重新评估：为什么单个查询需要超过堆内存的 60% 之多？

TIP

在 Fielddata 的大小 中，我们提过关于给 fielddata 的大小加一个限制，从而确保旧的无用 fielddata 被回收的方法。
indices.fielddata.cache.size 和 indices.breaker.fielddata.limit 之间的关系非常重要。如果断路器的限制低于缓存大小，没有数据会被回收。为了能正常工作，断路器的限制必须要比缓存大小要高。

值得注意的是：断路器是根据总堆内存大小估算查询大小的，而非根据实际堆内存的使用情况。这是由于各种技术原因造成的（例如，堆可能看上去是满的但实际上可能只是在等待垃圾回收，这使我们难以进行合理的估算）。但作为终端用户，这意味着设置需要保守，因为它是根据总堆内存必要的，而不是可用堆内存。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-07-22

- 1. Fielddata 的过滤

1. Fielddata 的过滤

设想我们正在运行一个网站允许用户收听他们喜欢的歌曲。为了让他们可以更容易的管理自己的音乐库，用户可以为歌曲设置任何他们喜欢的标签，这样我们就会有很多歌曲被附上 rock（摇滚）、hiphop（嘻哈）和 electronica（电音），但也会有些歌曲被附上 my_16th_birthday_favorite_anthem 这样的标签。

现在设想我们想要为用户展示每首歌曲最受欢迎的三个标签，很有可能 rock 这样的标签会排在三个中的最前面，而 my_16th_birthday_favorite_anthem 则不太可能得到评级。尽管如此，为了计算最受欢迎的标签，我们必须强制将这些一次性使用的项加载到内存中。

感谢 fielddata 过滤，我们可以控制这种状况。我们知道自己只对最流行的项感兴趣，所以我们可以简单地避免加载那些不太有意思的长尾项：

```
PUT /music/_mapping/song
{
  "properties": {
    "tag": {
      "type": "string",
      "fielddata": {
        "filter": {
          "frequency": {
            "min": 0.01,
            "min_segment_size": 500
          }
        }
      }
    }
  }
}
```

fielddata 关键字允许我们配置 fielddata 处理该字段的方式。frequency 过滤器允许我们基于项频率过滤加载 fielddata。只加载那些至少在本段文档中出现 1% 的项。忽略任何文档个数小于 500 的段。

有了这个映射，只有那些至少在本段文档中出现超过 1% 的项才会被加载到内存中。我们也可以指定一个最大词频，它可以被用来排除常用项，比如停用词。

这种情况下，词频是按照段来计算的。这是实现的一个限制：fielddata 是按段来加载的，所以可见的词频只是该段内的频率。但是，这个限制也有些有趣的特性：它可以让受欢迎的新项迅速提升到顶部。

比如一个新风格的歌曲在一夜之间受大众欢迎，我们可能想要将这种新风格的歌曲标签包括在最受欢迎列表中，但如果我们将依赖对索引做完整的计算获取词频，我们就必须等到新标签变得像 rock 和 electronica 一样流行。由于频度过滤的实现方式，新加的标签会很快作为高频标签出现在新段内，也当然会迅速上升到顶部。

min_segment_size 参数要求 Elasticsearch 忽略某个大小以下的段。如果一个段内只有少量文档，它的词频会非常粗略没有任何意义。小的分段会很快被合并到更大的分段中，某一刻超过这个限制，将会被纳入计算。

TIP

通过频次来过滤项并不是唯一的选择，我们也可以使用正则式来决定只加载那些匹配的项。例如，我们可以用 regex 过滤器 处理 twitte 上的消息只将以 # 号开始的标签加载到内存中。这假设我们使用的分析器会保留标点符号，像 whitespace 分析器。

Fielddata 过滤对内存使用有 巨大的 影响，权衡也是显而易见的：我们实际上是在忽略数据。但对于很多应用，这种权衡是合理的，因为这些数据根本就没有被使用到。内存的节省通常要比包括一个大量而无用的长尾项更为重要。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-22

- 1. 预加载fielddata
 - 1.1. 预加载 fielddata (Eagerly Loading Fielddata)
 - 1.2. 全局序号
 - 1.3. 构建全局序号 (Building global ordinals)
 - 1.4. 预构建全局序号 (Building global ordinals)
 - 1.5. 索引预热器 (Index Warmers)

1. 预加载fielddata

Elasticsearch 加载内存 fielddata 的默认行为是 延迟 加载。当 Elasticsearch 第一次查询某个字段时，它将会完整加载这个字段所有 Segment 中的倒排索引到内存中，以便于以后的查询能够获取更好的性能。

对于小索引段来说，这个过程的需要的时间可以忽略。但如果我们有一些 5 GB 的索引段，并希望加载 10 GB 的 fielddata 到内存中，这个过程可能会要数十秒。已经习惯亚秒响应的用户很难会接受停顿数秒卡着没反应的网站。

有三种方式可以解决这个延时高峰：

预加载 fielddata 预加载全局序号 缓存预热 所有的变化都基于同一概念：预加载 fielddata，这样在用户进行搜索时就不会碰到延迟高峰。

1.1. 预加载 fielddata (Eagerly Loading Fielddata)

第一个工具称为 预加载（与默认的 延迟加载相对）。随着新分段的创建（通过刷新、写入或合并等方式），启动字段预加载可以使那些对搜索不可见的分段里的 fielddata 提前 加载。

这就意味着首次命中分段的查询不需要促发 fielddata 的加载，因为 fielddata 已经被载入到内存。避免了用户遇到搜索卡顿的情形。

预加载是按字段启用的，所以我们可以控制具体哪个字段可以预先加载：

```
PUT /music/_mapping/_song
{
  "tags": {
    "type": "string",
    "fielddata": {
      "loading" : "eager"
    }
  }
}
```

设置 fielddata.loading: eager 可以告诉 Elasticsearch 预先将此字段的内容载入内存中

Fielddata 的载入可以使用 update-mapping API 对已有字段设置 lazy 或 eager 两种模式。



预加载只是简单的将载入 fielddata 的代价转移到索引刷新的时候，而不是查询时，从而大大提高了搜索体验。体积大的索引段会比体积小的索引段需要更长的刷新时间。通常，体积大的索引段是由那些已经对查询可见的小分段合并而成的，所以较慢的刷新时间也不是很重要。

1.2. 全局序号

有种可以用来降低字符串 fielddata 内存使用的技术叫做 序号 。设想我们有十亿文档，每个文档都有自己的 status 状态字段，状态总共有三种：status_pending、status_published、status_deleted 。如果我们为每个文档都保留其状态的完整字符串形式，那么每个文档就需要使用 14 到 16 字节，或总共 15 GB。

取而代之的是我们可以指定三个不同的字符串，对其排序、编号：0, 1, 2。

```
| Ordinal | Term | | ---- | ---- |
| 0 | status_deleted || 1 | status_pending || 2 | status_published |
序号字符串在序号列表中只存储一次，每个文档只要使用数值编号的序号来替代它原始的值。| Ordinal | Term | | ---- | ---- |
| 0 | 1 # pending || 1 | 1 # pending || 2 | 2 # published || 3 | 0 # deleted | 这样可以将内存使用从 15 GB 降到 1 GB 以下!
```

但这里有个问题，记得 fielddata 是按分段 来缓存的。如果一个分段只包含两个状态（status_deleted 和 status_published）。那么结果中的序号（0 和 1）就会与包含所有三个状态的分段不一样。

如果我们尝试对 status 字段运行 terms 聚合，我们需要对实际字符串的值进行聚合，也就是说我们需要识别所有分段中相同的值。一个简单粗暴的方式就是对每个分段执行聚合操作，返回每个分段的字符串值，再将它们归纳得出完整的结果。尽管这样做可行，但会很慢而且大量消耗 CPU。

取而代之的是使用一个被称为 全局序号 的结构。全局序号是一个构建在 fielddata 之上的数据结构，它只占用少量内存。唯一值是 跨所有分段 识别的，然后将它们存入一个序号列表中，正如我们描述过的那样。

现在， terms 聚合可以对全局序号进行聚合操作，将序号转换成真实字符串值的过程只会在聚合结束时发生一次。这会将聚合（和排序）的性能提高三到四倍。

1.3. 构建全局序号（Building global ordinals）

当然，天下没有免费的晚餐。全局序号分布在索引的所有段中，所以如果新增或删除一个分段时，需要对全局序号进行重建。重建需要读取每个分段的每个唯一项，基数越高（即存在更多的唯一项）这个过程会越长。

全局序号是构建在内存 fielddata 和 doc values 之上的。实际上，它们正是 doc values 性能表现不错的一个主要原因。

和 fielddata 加载一样，全局序号默认也是延迟构建的。首个需要访问索引内 fielddata 的请求会促发全局序号的构建。由于字段的基数不同，这会导致给用户带来显著延迟这一糟糕结果。一旦全局序号发生重建，仍会使用旧的全局序号，直到索引中的分段产生变化：在刷新、写入或合并之后。

1.4. 预构建全局序号（Building global ordinals）

单个字符串字段可以通过配置预先构建全局序号：

```
PUT /music/_mapping/_song
{
  "song_title": {
    "type": "string",
    "fielddata": {
      "loading": "eager_global_ordinals"
    }
  }
}
```

设置 eager_global_ordinals 也暗示着 fielddata 是预加载的。

正如 fielddata 的预加载一样，预构建全局序号发生在新分段对于搜索可见之前。

注意

序号的构建只被应用于字符串。数值信息（integers（整数）、geopoints（地理经纬度）、dates（日期）等等）不需要使用序号映射，因为这些值自己本质上就是序号映射。因此，我们只能为字符串字段预构建其全局序号。

也可以对 Doc values 进行全局序号预构建：

```
PUT /music/_mapping/_song
{
  "song_title": {
    "type": "string",
    "doc_values": true,
    "fielddata": {
      "loading": "eager_global_ordinals"
    }
  }
}
```

这种情况下，fielddata 没有载入到内存中，而是 doc values 被载入到文件系统缓存中。

与 fielddata 预加载不一样，预建全局序号会对数据的实时性产生影响，构建一个高基数的全局序号会使一个刷新延时数秒。选择在于是每次刷新时付出代价，还是在刷新后的第一次查询时。如果经常索引而查询较少，那么在查询时付出代价要比每次刷新时要好。如果写大于读，那么在选择在查询时重建全局序号将会是一个更好的选择。

Tip

针对实际场景优化全局序号的重建频次。如果我们有高基数字段需要花数秒钟重建，增加 refresh_interval 的刷新的时间从而可以使我们的全局序号保留更长的有效期，这也会节省 CPU 资源，因为我们重建的频次下降了。

1.5. 索引预热器 (Index Warmers)

最后我们谈谈 索引预热器。预热器早于 fielddata 预加载和全局序号预加载之前出现，它们仍然有其存在的理由。一个索引预热器允许我们指定一个查询和聚合须要在新分片对于搜索可见之前执行。这个想法是通过预先填充或 预热缓存 让用户永远无法遇到延迟的波峰。

原来，预热器最重要的用法是确保 fielddata 被预先加载，因为这通常是最耗时的一步。现在可以通过前面讨论的那些技术来更好的控制它，但是预热器还是可以用来预建过滤器缓存，当然我们也还是能选择用它来预加载 fielddata。

让我们注册一个预热器然后解释发生了什么：

```
PUT /music/_warmer/warmer_1
{
  "query": {
    "bool": {
      "filter": {
        "bool": {
          "should": [
            { "term": { "tag": "rock" } },
            { "term": { "tag": "hiphop" } },
            { "term": { "tag": "electronics" } }
          ]
        }
      }
    }
  },
  "aggs": {
    "price": {
      "histogram": {
        "field": "price",
        "interval": 10
      }
    }
  }
};
```

预热器被关联到索引（ music ）上，使用接入口 _warmer 以及 ID （ warmer_1 ）。

为三种最受欢迎的曲风预建过滤器缓存。

字段 price 的 fielddata 和全局序号会被预加载。

预热器是根据具体索引注册的，每个预热器都有唯一的 ID ，因为每个索引可能有多个预热器。

然后我们可以指定查询，任何查询。它可以包括查询、过滤器、聚合、排序值、脚本，任何有效的查询表达式都毫不夸张。这里的目的是想注册那些可以代表用户产生流量压力的查询，从而将合适的内容载入缓存。

当新建一个分段时， Elasticsearch 将会执行注册在预热器中的查询。执行这些查询会强制加载缓存， 只有在所有预热器执行完， 这个分段才会对搜索可见。



与预加载类似， 预热器只是将冷缓存的代价转移到刷新的时候。当注册预热器时， 做出明智的决定十分重要。为了确保每个缓存都被读入， 我们可以加入上千的预热器， 但这也会使新分段对于搜索可见的时间急剧上升。实际中， 我们会选择少量代表大多数用户的查询， 然后注册它们。

有些管理的细节（比如获得已有预热器和删除预热器）没有在本小节提到， 剩下的详细内容可以参考 [预热器文档（warmers documentation）](#)。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-22

- [1. 优化聚合查询](#)
 - [1.1. 深度优先与广度优先](#)

1. 优化聚合查询

“elasticsearch 里面桶的叫法和 SQL 里面分组的概念是类似的，一个桶就类似 SQL 里面的一个 group，多级嵌套的 aggregation，类似 SQL 里面的多字段分组 (group by field1,field2,) ，注意这里仅仅是概念类似，底层的实现原理是不一样的。 –译者注”

terms 桶基于我们的数据动态构建桶；它并不知道到底生成了多少桶。大多数时候对单个字段的聚合查询还是非常快的，但是当需要同时聚合多个字段时，就可能会产生大量的分组，最终结果就是占用 es 大量内存，从而导致 OOM 的情况发生。

假设我们现在有一些关于电影的数据集，每条数据里面会有一个数组类型的字段存储表演该电影的所有演员的名字。

```
{
  "actors" : [
    "Fred Jones",
    "Mary Jane",
    "Elizabeth Worthing"
  ]
}
```

如果我们想要查询出演影片最多的十个演员以及与他们合作最多的演员，使用聚合是非常简单的：

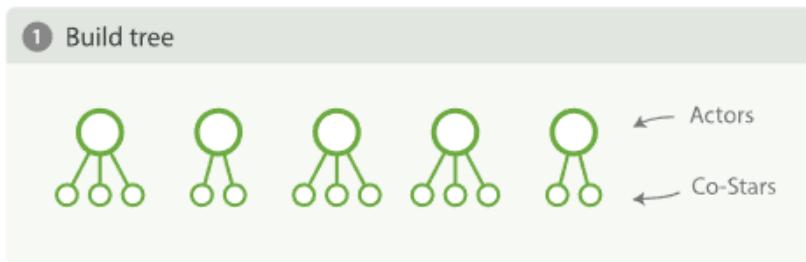
```
{
  "aggs" : {
    "actors" : {
      "terms" : {
        "field" : "actors",
        "size" : 10
      },
      "aggs" : {
        "costars" : {
          "terms" : {
            "field" : "actors",
            "size" : 5
          }
        }
      }
    }
  }
}
```

这会返回前十位出演最多的演员，以及与他们合作最多的五位演员。这看起来是一个简单的聚合查询，最终只返回 50 条数据！

但是，这个看上去简单的查询可以轻而易举地消耗大量内存，我们可以通过在内存中构建一个树来查看这个 terms 聚合。actors 聚合会构建树的第一层，每个演员都有一个桶。然后，内套在第一层的每个节点之下，costar 聚合会构建第二层，每

个联合出演一个桶, 请参见 Figure 42, “Build full depth tree” 所示。这意味着每部影片会生成 n^2 个桶!

Figure 42. Build full depth tree



用真实点的数据, 设想平均每部影片有 10 名演员, 每部影片就会生成 $10^2 = 100$ 个桶。如果总共有 20,000 部影片, 粗率计算就会生成 2,000,000 个桶。

现在, 记住, 聚合只是简单的希望得到前十位演员和与他们联合出演者, 总共 50 条数据。为了得到最终的结果, 我们创建了一个有 2,000,000 桶的树, 然后对其进行排序, 取 top10。图 Figure 43, “Sort tree” 和图 Figure 44, “Prune tree” 对这个过程进行了阐述。

Figure 43. Sort tree

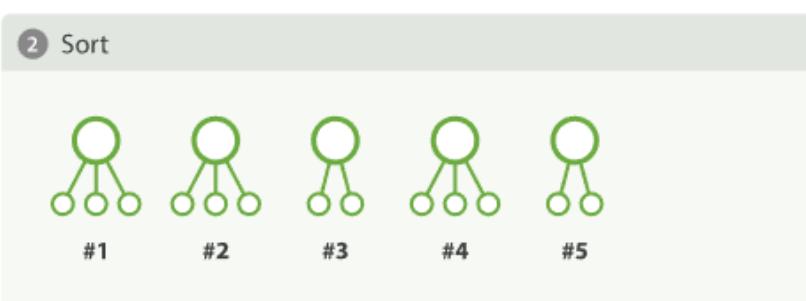
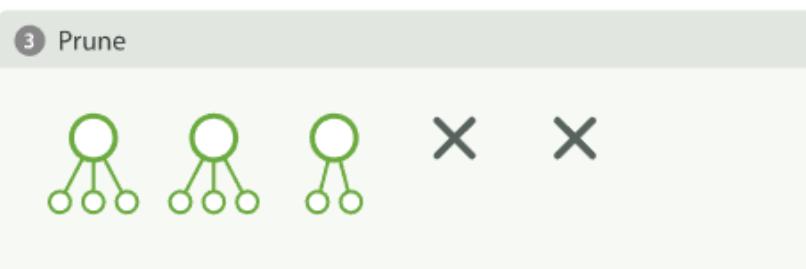


Figure 44. Prune tree



这时我们一定非常抓狂, 在 2 万条数据下执行任何聚合查询都是毫无压力的。如果我们有 2 亿文档, 想要得到前 100 位演员以及与他们合作最多的 20 位演员, 作为查询的最终结果会出现什么情况呢?

可以推测聚合出来的分组数非常大, 会使这种策略难以维持。世界上并不存在足够的内存来支持这种不受控制的聚合查询。

1.1. 深度优先与广度优先

Elasticsearch 允许我们改变聚合的 集合模式，就是为了应对这种状况。我们之前展示的策略叫做 深度优先，它是默认设置，先构建完整的树，然后修剪无用节点。深度优先 的方式对于大多数聚合都能正常工作，但对于如我们演员和联合演员这样例子的情形就不太适用。

为了应对这些特殊的应用场景，我们应该使用另一种集合策略叫做 广度优先。这种策略的工作方式有些不同，它先执行第一层聚合，再继续下一层聚合之前会先做修剪。图 Figure 45, “Build first level” 和图 Figure 47, “Prune first level” 对这个过程进行了阐述。

在我们的示例中，actors 聚合会首先执行，在这个时候，我们的树只有一层，但我们已经知道了前 10 位的演员！这就没有必要保留其他的演员信息，因为它们无论如何都不会出现在前十位中。

Figure 45. Build first level



Figure 46. Sort first level

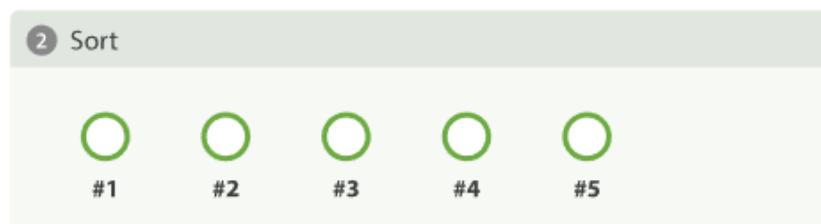
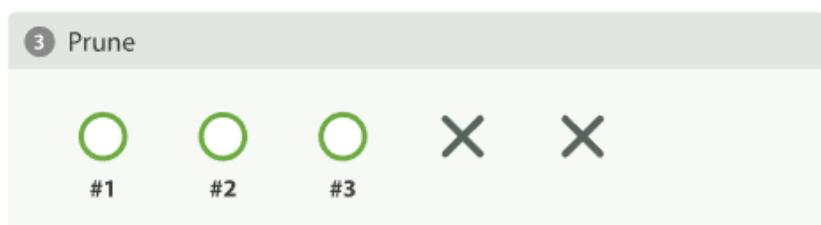


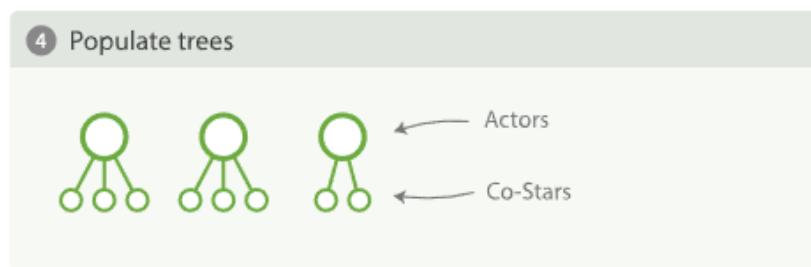
Figure 47. Prune first level



因为我们已经知道了前十名演员，我们可以安全的修剪其他节点。修剪后，下一层是基于它的 执行模式读入的，重复执行这个过程直到聚合完成，如图 Figure 48, “Populate full depth for remaining nodes” 所示。这种场景下，广度优先可以大幅

度节省内存。

Figure 48. Populate full depth for remaining nodes



要使用广度优先，只需简单的通过参数 `collect` 开启：

```
{
  "aggs" : {
    "actors" : {
      "terms" : {
        "field" : "actors",
        "size" : 10,
        "collect_mode" : "breadth_first"
      },
      "aggs" : {
        "costars" : {
          "terms" : {
            "field" : "actors",
            "size" : 5
          }
        }
      }
    }
  }
}
```

按聚合来开启 `breadth_first`。

广度优先仅仅适用于每个组的聚合数量远远小于当前总组数的情况下，因为广度优先会在内存中缓存裁剪后的仅仅需要缓存的每个组的所有数据，以便于它的子聚合分组查询可以复用上级聚合的数据。

广度优先的内存使用情况与裁剪后的缓存分组数据量是成线性的。对于很多聚合来说，每个桶内的文档数量是相当大的。想象一种按月分组的直方图，总组数肯定是固定的，因为每年只有12个月，这个时候每个月下的数据量可能非常大。这使广度优先不是一个好的选择，这也是为什么深度优先作为默认策略的原因。

针对上面演员的例子，如果数据量越大，那么默认的使用深度优先的聚合模式生成的总分组数就会非常多，但是预估二级的聚合字段分组后的数据量相比总的分组数会小很多所以这种情况下使用广度优先的模式能大大节省内存，从而通过优化聚合模式来大大提高了在某些特定场景下聚合查询的成功率。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-07-22

- 1. 深入搜索

1. 深入搜索

在基础入门中涵盖了基本工具并对它们有足够详细的描述，这让我们能够开始用 Elasticsearch 搜索数据。用不了多长时间，就会发现我们想要的更多：希望查询匹配更灵活，排名结果更精确，不同问题域下搜索更具体。想要进阶，只知道如何使用 match 查询是不够的，我们需要理解数据以及如何能够搜索到它们。本章会解释如何索引和查询我们的数据让我们能利用词的相似度（word proximity）、部分匹配（partial matching）、模糊匹配（fuzzy matching）以及语言感知（language awareness）这些优势。理解每个查询如何贡献相关度评分 _score 有助于调试我们的查询：确保我们认为的最佳匹配文档出现在结果首页，以及削减结果中几乎不相关的“长尾（long tail）”。搜索不仅仅是全文搜索：我们很大一部分数据都是结构化的，如日期和数字。我们会以说明结构化搜索与全文搜索最高效的结合方式开始本章的内容。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-07-22

- 1. 结构化搜索

1. 结构化搜索

结构化搜索（Structured search）是指有关探询那些具有内在结构数据的过程。比如日期、时间和数字都是结构化的：它们有精确的格式，我们可以对这些格式进行逻辑操作。比较常见的操作包括比较数字或时间的范围，或判定两个值的大小。

文本也可以是结构化的。如彩色笔可以有离散的颜色集合：红（red）、绿（green）、蓝（blue）。一个博客可能被标记了关键词 分布式（distributed）和 搜索（search）。电商网站上的商品都有 UPCs（通用产品码 Universal Product Codes）或其他的唯一标识，它们都需要遵从严格规定的、结构化的格式。

在结构化查询中，我们得到的结果总是非是即否，要么存于集合之中，要么存在集合之外。结构化查询不关心文件的相关度或评分；它简单的对文档包括或排除处理。

这在逻辑上是能说通的，因为一个数字不能比其他数字更适合存于某个相同范围。结果只能是：存于范围之中，抑或反之。同样，对于结构化文本来说，一个值要么相等，要么不等。没有更似这种概念。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-07-25

- 1. 控制相关度

1. 控制相关度

处理结构化数据（比如：时间、数字、字符串、枚举）的数据库，只需检查文档（或关系数据库里的行）是否与查询匹配。

布尔的是/非匹配是全文搜索的基础，但不止如此，我们还要知道每个文档与查询的相关度，在全文搜索引擎中不仅需要找到匹配的文档，还需根据它们相关度的高低进行排序。

全文相关的公式或相似算法（similarity algorithms）会将多个因素合并起来，为每个文档生成一个相关度评分 `_score`。本章中，我们会验证各种可变部分，然后讨论如何来控制它们。

当然，相关度不只与全文查询有关，也需要将结构化的数据考虑其中。可能我们正在找一个度假屋，需要一些的详细特征（空调、海景、免费 WiFi），匹配的特征越多相关度越高。可能我们还希望有一些其他的考虑因素，如回头率、价格、受欢迎度或距离，当然也同时考虑全文查询的相关度。

所有的这些都可以通过 Elasticsearch 强大的评分基础来实现。

本章会先从理论上介绍 Lucene 是如何计算相关度的，然后通过实际例子说明如何控制相关度的计算过程。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-22

- [1. 相关度评分背后的理论](#)
 - [1.1. 布尔模型](#)
 - [1.2. 词频/逆向文档频率](#)
 - [1.3. 词频](#)
 - [1.4. 逆向文档频率](#)
 - [1.5. 字段长度归一值](#)
 - [1.6. 结合使用](#)
- [2. 向量空间模型](#)

1. 相关度评分背后的理论

Lucene（或 Elasticsearch）使用布尔模型（Boolean model）查找匹配文档，并用一个名为 实用评分函数（practical scoring function）的公式来计算相关度。这个公式借鉴了词频/逆向文档频率（term frequency/inverse document frequency）和向量空间模型（vector space model），同时也加入了一些现代的新特性，如协调因子（coordination factor），字段长度归一化（field length normalization），以及词或查询语句权重提升。

注意

不要紧张！这些概念并没有像它们字面看起来那么复杂，尽管本小节提到了算法、公式和数学模型，但内容还是让人容易理解的，与理解算法本身相比，了解这些因素如何影响结果更为重要。

1.1. 布尔模型

布尔模型（Boolean Model）只是在查询中使用 AND、OR 和 NOT（与、或和非）这样的条件来查找匹配的文档，以下查询：

full AND text AND search AND (elasticsearch OR lucene) 会将所有包括词 full、text 和 search，以及 elasticsearch 或 lucene 的文档作为结果集。

这个过程简单且快速，它将所有可能不匹配的文档排除在外。

1.2. 词频/逆向文档频率

当匹配到一组文档后，需要根据相关度排序这些文档，不是所有的文档都包含所有词，有些词比其他的词更重要。一个文档的相关度评分部分取决于每个查询词在文档中的权重。

词的权重由三个因素决定，在 什么是相关 中已经有所介绍，有兴趣可以了解下面的公式，但并不要求记住。

1.3. 词频

词在文档中出现的频度是多少？频度越高，权重越高。5 次提到同一词的字段比只提到 1 次的更相关。词频的计算方式如下：

$$tf(t \text{ in } d) = \sqrt{frequency}$$

词 t 在文档 d 的词频 (tf) 是该词在文档中出现次数的平方根。

如果不在意词在某个字段中出现的频次，而只在意是否出现过，则可以在字段映射中禁用词频统计：

```
PUT /my_index
{
  "mappings": {
    "doc": {
      "properties": {
        "text": {
          "type": "string",
          "index_options": "docs"
        }
      }
    }
  }
}
```

将参数 index_options 设置为 docs 可以禁用词频统计及词频位置，这个映射的字段不会计算词的出现次数，对于短语或近似查询也不可用。要求精确查询的 not_analyzed 字符串字段会默认使用该设置。

1.4. 逆向文档频率

词在集合所有文档里出现的频率是多少？频次越高，权重越低。常用词如 and 或 the 对相关度贡献很少，因为它们在多数文档中都会出现，一些不常见词如 elastic 或 hippopotamus 可以帮助我们快速缩小范围找到感兴趣的文档。逆向文档频率的计算公式如下：

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

词 t 的逆向文档频率 (idf) 是：索引中文档数量除以所有包含该词的文档数，然后求其对数。

1.5. 字段长度归一值

字段的长度是多少？字段越短，字段的权重越高。如果词出现在类似标题 title 这样的字段，要比它出现在内容 body 这样的字段中的相关度更高。字段长度的归一值公式如下：

$$\text{norm}(d) = 1 / \sqrt{\text{numTerms}}$$

字段长度归一值 (norm) 是字段中词数平方根的倒数。

字段长度的归一值对全文搜索非常重要，许多其他字段不需要有归一值。无论文档是否包括这个字段，索引中每个文档的每个 string 字段都大约占用 1 个 byte 的空间。对于 not_analyzed 字符串字段的归一值默认是禁用的，而对于 analyzed 字段也可以通过修改字段映射禁用归一值：

```
PUT /my_index
{
  "mappings": {
    "doc": {
      "properties": {
        "text": {
          "type": "string",
          "norms": { "enabled": false }
        }
      }
    }
  }
}
```

这个字段不会将字段长度归一值考虑在内，长字段和短字段会以相同长度计算评分。

对于有些应用场景如日志，归一值不是很有用，要关心的只是字段是否包含特殊的错误码或者特定的浏览器唯一标识符。字段的长度对结果没有影响，禁用归一值可以节省大量内存空间。

1.6. 结合使用

以下三个因素——词频（term frequency）、逆向文档频率（inverse document frequency）和字段长度归一值（field-length norm）——是在索引时计算并存储的。最后将它们结合在一起计算单个词在特定文档中的权重。

Tip

前面公式中提到的 文档 实际上是指文档里的某个字段，每个字段都有它自己的倒排索引，因此字段的 TF/IDF 值就是文档的 TF/IDF 值。

当用 explain 查看一个简单的 term 查询时（参见 explain），可以发现与计算相关度评分的因子就是前面章节介绍的这些：

```
PUT /my_index/doc/1
{ "text" : "quick brown fox" }

GET /my_index/doc/_search?explain
{
  "query": {
    "term": {
      "text": "fox"
    }
  }
}
```

以上请求（简化）的 explanation 解释如下：

weight(text:fox in 0) [PerFieldSimilarity]: 0.15342641 result of: fieldWeight in 0
 0.15342641 product of: tf(freq=1.0), with freq of 1: 1.0 idf(docFreq=1,
 maxDocs=1): 0.30685282 fieldNorm(doc=0): 0.5

词 fox 在文档的内部 Lucene doc ID 为 0，字段是 text 里的最终评分。

词 fox 在该文档 text 字段中只出现了一次。

fox 在所有文档 text 字段索引的逆向文档频率。

该字段的字段长度归一值。

当然，查询通常不止一个词，所以需要一种合并多词权重的方式——向量空间模型（vector space model）。

2. 向量空间模型

向量空间模型（vector space model）提供一种比较多词查询的方式，单个评分代表文档与查询的匹配程度，为了做到这点，这个模型将文档和查询都以向量（vectors）的形式表示：

向量实际上就是包含多个数的一维数组，例如：

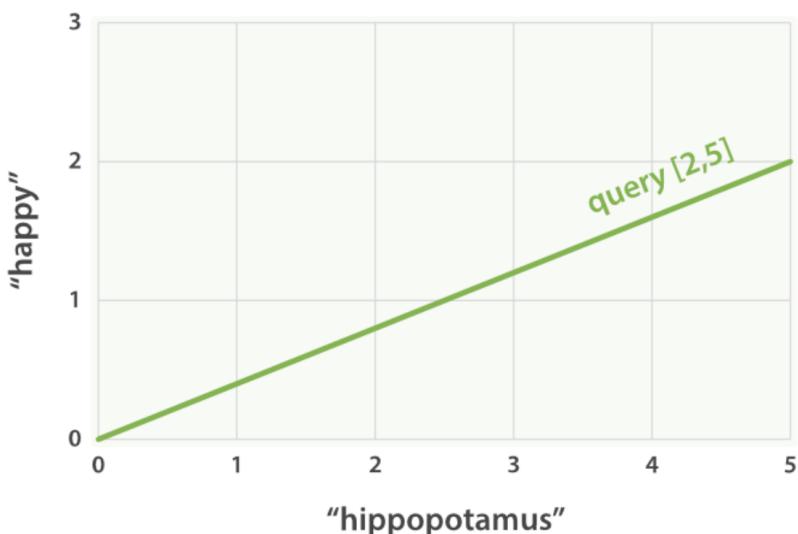
[1,2,5,22,3,8] 在向量空间模型里，向量空间模型里的每个数字都代表一个词的权重，与词频/逆向文档频率（term frequency/inverse document frequency）计算方式类似。

Tip

尽管 TF/IDF 是向量空间模型计算词权重的默认方式，但不是唯一方式。Elasticsearch 还有其他模型如 Okapi-BM25。TF/IDF 是默认的因为它是个经检验过的简单又高效的算法，可以提供高质量的搜索结果。

设想如果查询“happy hippopotamus”，常见词 happy 的权重较低，不常见词 hippopotamus 权重较高，假设 happy 的权重是 2，hippopotamus 的权重是 5，可以将这个二维向量——[2,5]——在坐标系下作条直线，线的起点是 (0,0) 终点是 (2,5)，如图 Figure 27，“表示“happy hippopotamus”的二维查询向量”。

Figure 27. 表示“happy hippopotamus”的二维查询向量



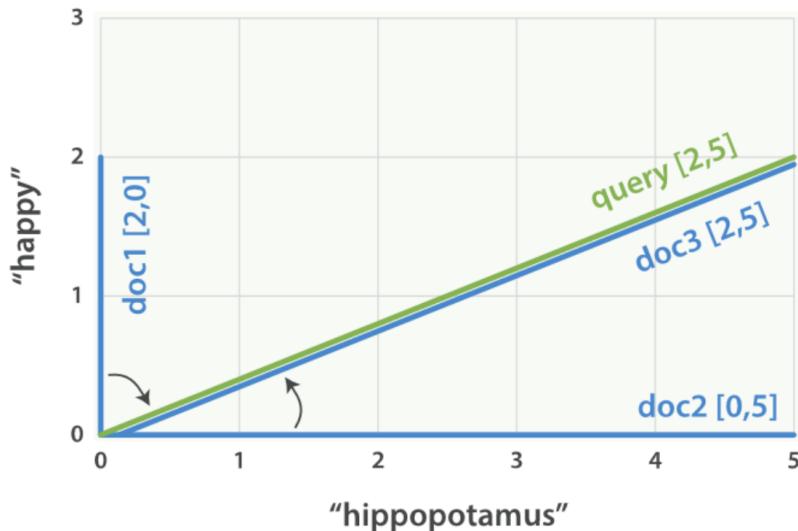
现在，设想我们有三个文档：

I am happy in summer。After Christmas I'm a hippopotamus。The happy hippopotamus helped Harry。可以为每个文档都创建包括每个查询词——happy 和 hippopotamus——权重的向量，然后将这些向量置入同一个坐标系中，如图

Figure 28, ““happy hippopotamus” 查询及文档向量”：

文档 1: (happy,_) —— [2,0] 文档 2: (_ ,hippopotamus) —— [0,5] 文档 3:
(happy,hippopotamus) —— [2,5]

Figure 28. “happy hippopotamus” 查询及文档向量



向量之间是可以比较的，只要测量查询向量和文档向量之间的角度就可以得到每个文档的相关度，文档 1 与查询之间的角度最大，所以相关度低；文档 2 与查询间的角度较小，所以更相关；文档 3 与查询的角度正好吻合，完全匹配。

Tip

在实际中，只有二维向量（两个词的查询）可以在平面上表示，幸运的是，线性代数——作为数学中处理向量的一个分支——为我们提供了计算两个多维向量间角度工具，这意味着可以使用如上同样的方式来解释多个词的查询。

关于比较两个向量的更多信息可以参考 [余弦近似度 \(cosine similarity\)](#)。

现在已经讲完评分计算的基本理论，我们可以继续了解 Lucene 是如何实现评分计算的。

Copyright © Songbai Yang all right reserved, powered by Gitbook 该文件修订时间：2021-07-23

- [1. Lucene 的实用评分函数](#)
 - [1.1. 查询归一因子](#)
 - [1.2. 查询协调](#)
 - [1.3. 索引时字段权重提升](#)

1. Lucene 的实用评分函数

对于多词查询，Lucene 使用 布尔模型（Boolean model） 、 TF/IDF 以及 向量空间模型（vector space model），然后将它们组合到单个高效的包里以收集匹配文档并进行评分计算。

一个多词查询

```
GET /my_index/doc/_search
{
  "query": {
    "match": {
      "text": "quick fox"
    }
  }
}
```

会在内部被重写为：

```
GET /my_index/doc/_search
{
  "query": {
    "bool": {
      "should": [
        {"term": { "text": "quick" }},
        {"term": { "text": "fox" }}
      ]
    }
  }
}
```

bool 查询实现了布尔模型，在这个例子中，它会将包括词 quick 和 fox 或两者兼有的文档作为查询结果。

只要一个文档与查询匹配，Lucene 就会为查询计算评分，然后合并每个匹配词的评分结果。这里使用的评分计算公式叫做 实用评分函数（practical scoring function）。看似很高大上，但是别被吓到——多数的组件都已经介绍过，下一步会讨论它引入的一些新元素。

$\text{score}(q, d) = (1) \cdot \text{queryNorm}(q) \cdot \sum_{t \in q} \text{coord}(q, d) \cdot \sum_{t \in d} (\text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot \text{t.getBoost()} \cdot \text{norm}(t, d))$

(1) $\text{score}(q, d)$ 是文档 d 与查询 q 的相关度评分。

(2) $\text{queryNorm}(q)$ 是 查询归一化 因子（新）。

(3) $\text{coord}(q, d)$ 是 协调 因子（新）。

(4) 查询 q 中每个词 t 对于文档 d 的权重和。

(5)tf(t in d) 是词 t 在文档 d 中的 词频 。

(6)idf(t) 是词 t 的 逆向文档频率 。

(7)t.getBoost() 是查询中使用的 boost (新) 。

norm(t,d) 是 字段长度归一值，与 索引时字段层 boost (如果存在) 的和 (新) 。

上节已介绍过 score 、 tf 和 idf 。现在来介绍 queryNorm 、 coord 、 t.getBoost 和 norm 。我们会在本章后面继续探讨 查询时的权重提升 的问题，但是首先需要了解查询归一化、协调和索引时字段层面的权重提升等概念。

1.1. 查询归一因子

查询归一因子 (queryNorm) 试图将查询 归一化，这样就能将两个不同的查询结果相比较。

Tip

尽管查询归一值的目的是为了使查询结果之间能够相互比较，但是它并不十分有效，因为相关度评分 _score 的目的是为了将当前查询的结果进行排序，比较不同查询结果的相关度评分没有太大意义。

这个因子是在查询过程的最前面计算的，具体的计算依赖于具体查询，一个典型的实现如下：

$\text{queryNorm} = 1 / \sqrt{\text{sumOfSquaredWeights}}$

sumOfSquaredWeights 是查询里每个词的 IDF 的平方和。

Tip

相同查询归一化因子会被应用到每个文档，不能被更改，总而言之，可以被忽略。

1.2. 查询协调

协调因子 (coord) 可以为那些查询词包含度高的文档提供奖励，文档里出现的查询词越多，它越有机会成为好的匹配结果。

设想查询 quick brown fox ，每个词的权重都是 1.5 。如果没有协调因子，最终评分会是文档里所有词权重的总和。例如：

- 文档里有 fox → 评分： 1.5
- 文档里有 quick fox → 评分： 3.0
- 文档里有 quick brown fox → 评分： 4.5

协调因子将评分与文档里匹配词的数量相乘，然后除以查询里所有词的数量，如果使用协调因子，评分会变成：

- 文档里有 fox → 评分： $1.5 * 1 / 3 = 0.5$
- 文档里有 quick fox → 评分： $3.0 * 2 / 3 = 2.0$
- 文档里有 quick brown fox → 评分： $4.5 * 3 / 3 = 4.5$

协调因子能使包含所有三个词的文档比只包含两个词的文档评分要高出很多。

回想将查询 quick brown fox 重写成 bool 查询的形式：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "term": { "text": "quick" }},
        { "term": { "text": "brown" }},
        { "term": { "text": "fox"   }}
      ]
    }
  }
}
```

bool 查询默认会对所有 should 语句使用协调功能，不过也可以将其禁用。为什么要这样做？通常的回答是——无须这样。查询协调通常是件好事，当使用 bool 查询将多个高级查询如 match 查询包裹的时候，让协调功能开启是有意义的，匹配的语句越多，查询请求与返回文档间的重叠度就越高。

但在某些高级应用中，将协调功能关闭可能更好。设想正在查找同义词 jump、leap 和 hop 时，并不关心会出现多少个同义词，因为它们都表示相同的意思，实际上，只有其中一个同义词会出现，这是不使用协调因子的一个好例子：

```
GET /_search
{
  "query": {
    "bool": {
      "disable_coord": true,
      "should": [
        { "term": { "text": "jump" }},
        { "term": { "text": "hop"  }},
        { "term": { "text": "leap" }}
      ]
    }
  }
}
```

当使用同义词的时候（参照：[同义词](#)），Lucene 内部是这样的：重写的查询会禁用同义词的协调功能。大多数禁用操作的应用场景是自动处理的，无须为此担心。

1.3. 索引时字段权重提升

我们会讨论查询时的权重提升，让字段 权重提升 就是让某个字段比其他字段更重要。当然在索引时也能做到如此。实际上，权重的提升会被应用到字段的每个词，而不是字段本身。

将提升值存储在索引中无须更多空间，这个字段层索引时的提升值与字段长度归一值（参见 [字段长度归一值](#)）一起作为单个字节存于索引， $\text{norm}(t, d)$ 是前面公式 的返回值。

⚠ 我们不建议在建立索引时对字段提升权重，有以下原因：

将提升值与字段长度归一值合在单个字节中存储会丢失字段长度归一值的精度，这样会导致 Elasticsearch 不知如何区分包含三个词的字段和包含五个词的字段。要想改变索引时的提升值，就必须重新为所有文档建立索引，与此不同的是，查询时的提升值可以随着每次查询的不同而更改。如果一个索引时权重提升的字段有多个值，提升值会按照每个值来自乘，这会导致该字段的权重急剧上升。

查询时赋予权重是更为简单、清楚、灵活的选择。

了解了查询归一化、协同和索引时权重提升这些方式后，可以进一步了解相关度计算最有用的工具：查询时的权重提升。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-07-23

- 1. 查询时权重提升
 - 1.1. 提升权重
 - 1.2. t.getBoost()

1. 查询时权重提升

在语句优先级 (Prioritizing Clauses) 中，我们解释过如何在搜索时使用 boost 参数让一个查询语句比其他语句更重要。例如：

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

title 查询语句的重要性是 content 查询的 2 倍，因为它的权重提升值为 2。

没有设置 boost 的查询语句的值为 1。

查询时的权重提升 是可以用来影响相关度的主要工具，任意类型的查询都能接受 boost 参数。将 boost 设置为 2，并不代表最终的评分 _score 是原值的两倍；实际的权重值会经过归一化和一些其他内部优化过程。尽管如此，它确实想要表明一个提升值为 2 的句子的重要性是提升值为 1 语句的两倍。

在实际应用中，无法通过简单的公式得出某个特定查询语句的“正确”权重提升值，只能通过不断尝试获得。需要记住的是 boost 只是影响相关度评分的其中一个因子；它还需要与其他因子相互竞争。在前例中，title 字段相对 content 字段可能已经有一个“缺省的”权重提升值，这因为在字段长度归一值中，标题往往比相关内容要短，所以不要想当然的去盲目提升一些字段的权重。选择权重，检查结果，如此反复。

1.1. 提升权重

当在多个索引中搜索时，可以使用参数 indices_boost 来提升整个索引的权重，在下面例子中，当要为最近索引的文档分配更高权重时，可以这么做：

```
GET /docs_2014_*/_search (1)
{
  "indices_boost": { (2)
    "docs_2014_10": 3,
    "docs_2014_09": 2
  },
  "query": {
    "match": {
      "text": "quick brown fox"
    }
  }
}
```

- (1) 这个多索引查询涵盖了所有以字符串 docs2014 开始的索引。
(2) 其中，索引 docs_2014_10 中的所有文件的权重是 3，索引 docs_2014_09 中是 2，其他所有匹配的索引权重为默认值 1。

1.2. t.getBoost()

这些提升值在 Lucene 的 实用评分函数 中可以通过 t.getBoost() 获得。权重提升不会被应用于它在查询表达式中出现的层，而是会被合并下转至每个词中。
t.getBoost() 始终返回当前词的权重或当前分析链上查询的权重。

TIP

实际上，要想解读 explain 的输出是相当复杂的，在 explanation 里面完全看不到 boost 值，也完全无法访问上面提到的 t.getBoost() 方法，权重值融合在 queryNorm 中并应用到每个词。尽管说，queryNorm 对于每个词都是相同的，还是会发现一个权重提升过的词的 queryNorm 值要高于一个没有提升过的。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-23

- 1. 使用查询结构修改相关度

1. 使用查询结构修改相关度

Elasticsearch 的查询表达式相当灵活，可以通过调整查询结构中查询语句的所处层次，从而或多或少改变其重要性，比如，设想下面这个查询：

quick OR brown OR red OR fox 可以将所有词都放在 bool 查询的同一层中：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "term": { "text": "quick" } },
        { "term": { "text": "brown" } },
        { "term": { "text": "red" } },
        { "term": { "text": "fox" } }
      ]
    }
  }
}
```

这个查询可能最终给包含 quick 、 red 和 brown 的文档评分与包含 quick 、 red 、 fox 文档的评分相同，这里 Red 和 brown 是同义词，可能只需要保留其中一个，而我们真正要表达的意思是想做以下查询：

quick OR (brown OR red) OR fox 根据标准的布尔逻辑，这与原始的查询是完全一样的，但是我们已经在 组合查询（Combining Queries） 中看到， bool 查询不关心文档匹配的程度，只关心是否能匹配。

上述查询有个更好的方式：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "term": { "text": "quick" } },
        { "term": { "text": "fox" } },
        {
          "bool": {
            "should": [
              { "term": { "text": "brown" } },
              { "term": { "text": "red" } }
            ]
          }
        }
      ]
    }
  }
}
```

现在， red 和 brown 处于相互竞争的层次， quick 、 fox 以及 red OR brown 则是处于顶层且相互竞争的词。

我们已经讨论过如何使用 `match`、`multi_match`、`term`、`bool` 和 `dis_max` 查询修改相关度评分。本章后面的内容会介绍另外三个与相关度评分有关的查询：`boosting` 查询、`constant_score` 查询和 `function_score` 查询。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-07-23

- [1. Not Quite Not](#)
 - [1.1. 权重提升查询](#)

1. Not Quite Not

在互联网上搜索“Apple”，返回的结果很可能是一个公司、水果和各种食谱。我们可以在 bool 查询中用 must_not 语句来排除像 pie、tart、crumble 和 tree 这样的词，从而将查询结果的范围缩小至只返回与“Apple”（苹果）公司相关的结果：

```
GET /_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "text": "apple"
          }
        }
      ],
      "must_not": [
        {
          "match": {
            "text": "pie tart fruit crumble tree"
          }
        }
      ]
    }
  }
}
```

但谁又敢保证在排除 tree 或 crumble 这种词后，不会错失一个与苹果公司特别相关的文档呢？有时，must_not 条件会过于严格。

1.1. 权重提升查询

boosting 查询 恰恰能解决这个问题。它仍然允许我们将关于水果或甜点的结果包括到结果中，但是使它们降级——即降低它们原来可能应有的排名：

```
GET /_search
{
  "query": {
    "boosting": {
      "positive": [
        {
          "match": {
            "text": "apple"
          }
        }
      ],
      "negative": [
        {
          "match": {
            "text": "pie tart fruit crumble tree"
          }
        }
      ],
      "negative_boost": 0.5
    }
  }
}
```

它接受 positive 和 negative 查询。只有那些匹配 positive 查询的文档罗列出来，对于那些同时匹配 negative 查询的文档将通过文档的原始 _score 与 negative_boost 相乘的方式降级后的结果。

为了达到效果，negative_boost 的值必须小于 1.0。在这个示例中，所有包含负向词的文档评分 _score 都会减半。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-07-23

- [1. 忽略 TF/IDF](#)
 - [1.1. constant_score](#)

1. 忽略 TF/IDF

有时候我们根本不关心 TF/IDF，只想知道一个词是否在某个字段中出现过。可能搜索一个度假屋并希望它能尽可能有以下设施：

- WiFi
- Garden (花园)
- Pool (游泳池)

这个度假屋的文档如下：

```
{ "description": "A delightful four-bedroomed house with ... " }
```

可以用简单的 match 查询进行匹配：

```
GET /_search
{
  "query": {
    "match": {
      "description": "wifi garden pool"
    }
  }
}
```

但这并不是真正的全文搜索，此种情况下，TF/IDF 并无用处。我们既不关心 wifi 是否为一个普通词，也不关心它在文档中出现是否频繁，关心的只是它是否曾出现过。实际上，我们希望根据房屋不同设施的数量对其进行排名——设施越多越好。如果设施出现，则记 1 分，不出现记 0 分。

1.1. constant_score

在 [constant_score](#) 查询中，它可以包含查询或过滤，为任意一个匹配的文档指定评分 1，忽略 TF/IDF 信息：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "constant_score": {
          "query": { "match": { "description": "wifi" } }
        }},
        { "constant_score": {
          "query": { "match": { "description": "garden" } }
        }},
        { "constant_score": {
          "query": { "match": { "description": "pool" } }
        }}
      ]
    }
  }
}
```

或许不是所有的设施都同等重要——对某些用户来说有些设施更有价值。如果最重要的设施是游泳池，那我们可以为更重要的设施增加权重：

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        { "constant_score": {
          "query": { "match": { "description": "wifi" } }
        }},
        { "constant_score": {
          "query": { "match": { "description": "garden" } }
        }},
        { "constant_score": {
          "boost": 2
          "query": { "match": { "description": "pool" } }
        }}
      ]
    }
  }
}
```

pool 语句的权重提升值为 2，而其他的语句为 1。

注意

最终的评分并不是所有匹配语句的简单求和，协调因子 (coordination factor) 和查询归一化因子 (query normalization factor) 仍然会被考虑在内。

我们可以给 features 字段加上 not_analyzed 类型来提升度假屋文档的匹配能力：

```
{ "features": [ "wifi", "pool", "garden" ] }
```

默认情况下，一个 not_analyzed 字段会禁用 字段长度归一值 (field-length norms) 的功能，并将 index_options 设为 docs 选项，禁用 词频，但还是存在问题：每个词的 倒排文档频率 仍然会被考虑。

可以采用与之前相同的方法 constant_score 查询来解决这个问题：

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

实际上，每个设施都应该看成一个过滤器，对于度假屋来说要么具有某个设施要么没有——过滤器因为其性质天然合适。而且，如果使用过滤器，我们还可以利用缓存。

这里的问题是：过滤器无法计算评分。这样就需要寻求一种方式将过滤器和查询间的差异抹平。`function_score` 查询不仅正好可以扮演这个角色，而且有更强大的功能。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-07-23

- [1. function_score 查询](#)

1. function_score 查询

`function_score` 查询 是用来控制评分过程的终极武器，它允许为每个与主查询匹配的文档应用一个函数，以达到改变甚至完全替换原始查询评分 `_score` 的目的。

实际上，也能用过滤器对结果的子集 应用不同的函数，这样一箭双雕：既能高效评分，又能利用过滤器缓存。

Elasticsearch 预定义了一些函数：

- `weight`

为每个文档应用一个简单而不被规范化的权重提升值：当 `weight` 为 2 时，最终结果为 $2 * \text{_score}$ 。

- `field_value_factor`

使用这个值来修改 `_score`，如将 `popularity` 或 `votes`（受欢迎或赞）作为考虑因素。

- `random_score`

为每个用户都使用一个不同的随机评分对结果排序，但对某一具体用户来说，看到的顺序始终是一致的。

- 衰减函数 —— `linear`、`exp`、`gauss`

将浮动值结合到评分 `_score` 中，例如结合 `publish_date` 获得最近发布的文档，结合 `geo_location` 获得更接近某个具体经纬度 (lat/lon) 地点的文档，结合 `price` 获得更接近某个特定价格的文档。

- `script_score`

如果需求超出以上范围时，用自定义脚本可以完全控制评分计算，实现所需逻辑。

如果没有 `function_score` 查询，就不能将全文查询与最新发生这种因子结合在一起评分，而不得不根据评分 `_score` 或时间 `date` 进行排序；这会相互影响抵消两种排序各自的效果。这个查询可以使两个效果融合：可以仍然根据全文相关度进行排序，但也会同时考虑最新发布文档、流行文档、或接近用户希望价格的产品。正如所设想的，查询要考虑所有这些因素会非常复杂，让我们先从简单的例子开始，然后顺着梯子慢慢向上爬，增加复杂度。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-07-24

- 1. 按受欢迎度提升权重
 - 1.1. modifier
 - 1.2. factor
 - 1.3. boost_mode
 - 1.4. max_boost

1. 按受欢迎度提升权重

设想有个网站供用户发布博客并且可以让他们为自己喜欢的博客点赞，我们希望将更受欢迎的博客放在搜索结果列表中相对较高的位置，同时全文搜索的评分仍然作为相关度的主要排序依据，可以简单的通过存储每个博客的点赞数来实现它：

```
PUT /blogposts/post/1
{
  "title": "About popularity",
  "content": "In this post we will talk about...",
  "votes": 6
}
```

在搜索时，可以将 function_score 查询与 field_value_factor 结合使用，即将点赞数与全文相关度评分结合：

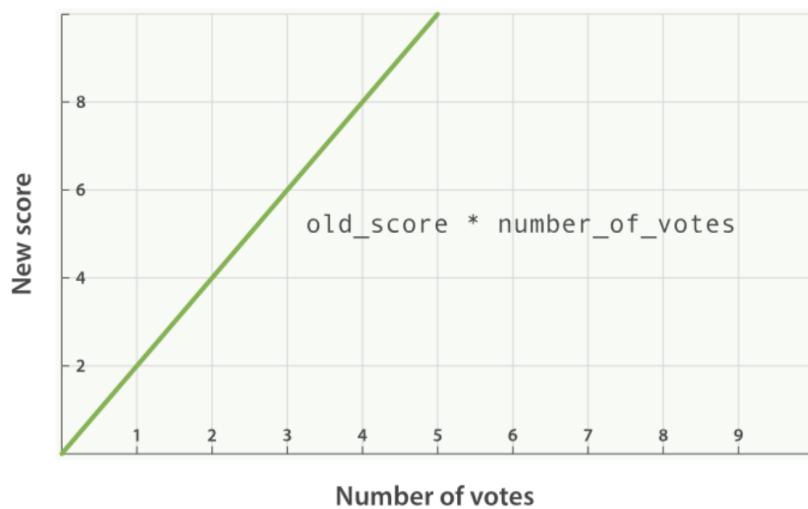
```
GET /blogposts/post/_search
{
  "query": {
    "function_score": { (1)
      "query": { (2)
        "multi_match": {
          "query": "popularity",
          "fields": [ "title", "content" ]
        }
      },
      "field_value_factor": { (3)
        "field": "votes" (4)
      }
    }
  }
}
```

- (1) function_score 查询将主查询和函数包括在内。
- (2) 主查询优先执行。
- (3) field_value_factor 函数会被应用到每个与主 query 匹配的文档。
- (4) 每个文档的 votes 字段都必须有值供 function_score 计算。如果没有文档的 votes 字段有值，那么就必须使用 missing 属性 提供的默认值来进行评分计算。

在前面示例中，每个文档的最终评分 _score 都做了如下修改：

`new_score = old_score * number_of_votes` 然而这并不会带来出人意料的好结果，全文评分 _score 通常处于 0 到 10 之间，如下图 Figure 29，“受欢迎度的线性关系基于 _score 的原始值 2.0” 中，有 10 个赞的博客会掩盖掉全文评分，而 0 个赞的博客的评分会被置为 0 。

Figure 29. 受欢迎度的线性关系基于 `_score` 的原始值 2.0



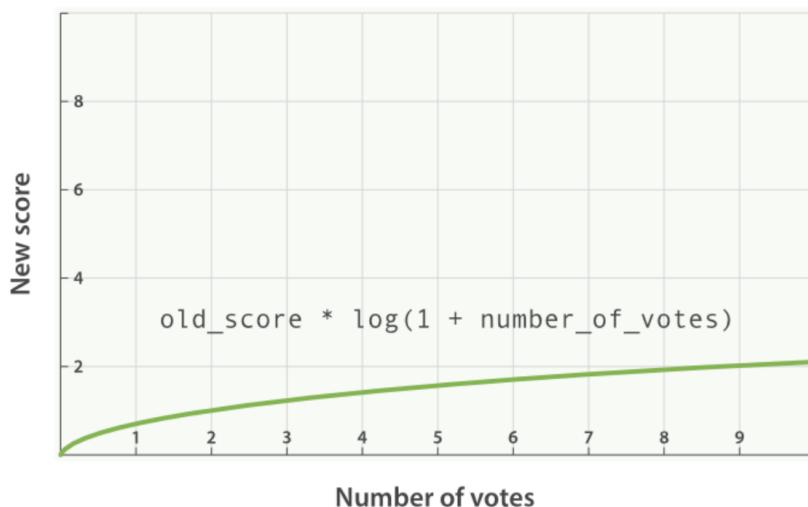
1.1. modifier

一种融入受欢迎度更好方式是用 modifier 平滑 votes 的值。换句话说，我们希望最开始的一些赞更重要，但是其重要性会随着数字的增加而降低。0 个赞与 1 个赞的区别应该比 10 个赞与 11 个赞的区别大很多。

对于上述情况，典型的 modifier 应用是使用 `log1p` 参数值，公式如下：

`new_score = old_score * log(1 + number_of_votes)` log 对数函数使 votes 赞字段的评分曲线更平滑，如图 Figure 30，“受欢迎度的对数关系基于 `_score` 的原始值 2.0”：

Figure 30. 受欢迎度的对数关系基于 `_score` 的原始值 2.0



带 modifier 参数的请求如下：

```
GET /blogposts/post/_search
{
  "query": {
    "function_score": {
      "query": {
        "multi_match": {
          "query": "popularity",
          "fields": [ "title", "content" ]
        }
      },
      "field_value_factor": {
        "field": "votes",
        "modifier": "log1p" (1)
      }
    }
  }
}
```

(1) modifier 为 log1p。

修饰语 modifier 的值可以为: none (默认状态)、log、log1p、log2p、ln、ln1p、ln2p、square、sqrt 以及 reciprocal。想要了解更多信息请参照:
[field_value_factor 文档](#).

1.2. factor

可以通过将 votes 字段与 factor 的积来调节受欢迎程度效果的高低:

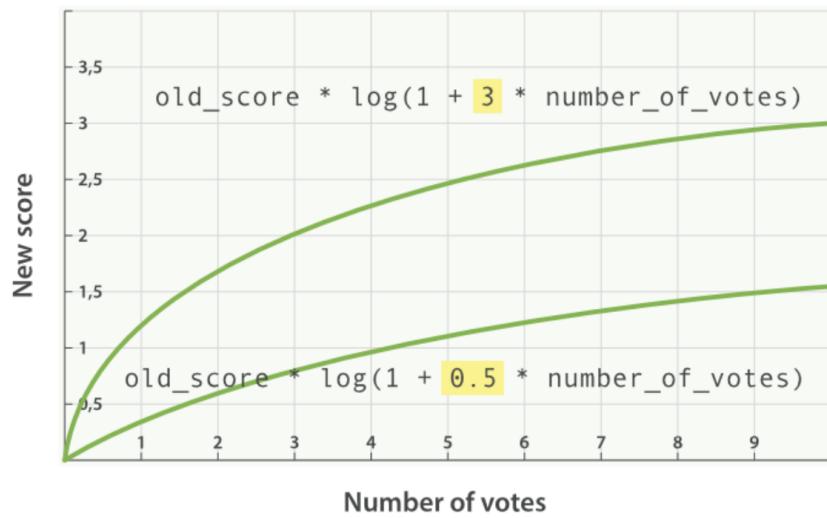
```
GET /blogposts/post/_search
{
  "query": {
    "function_score": {
      "query": {
        "multi_match": {
          "query": "popularity",
          "fields": [ "title", "content" ]
        }
      },
      "field_value_factor": {
        "field": "votes",
        "modifier": "log1p",
        "factor": 2 (1)
      }
    }
  }
}
```

(1) 双倍效果。

添加了 factor 会使公式变成这样:

$\text{new_score} = \text{old_score} \log(1 + \text{factor} \text{ number_of_votes})$

factor 值大于 1 会提升效果, factor 值小于 1 会降低效果, 如图 Figure 31, “受欢迎度的对数关系基于多个不同因子”。

Figure 31. 受欢迎度的对数关系基于多个不同因子

1.3. boost_mode

或许将全文评分与 field_value_factor 函数值乘积的效果仍然可能太大， 我们可以通过参数 boost_mode 来控制函数与查询评分 _score 合并后的结果， 参数接受的值为：

- multiply
评分 _score 与函数值的积（默认）
- sum
评分 _score 与函数值的和
- min
评分 _score 与函数值间的较小值
- max
评分 _score 与函数值间的较大值
- replace
函数值替代评分 _score

与使用乘积的方式相比， 使用评分 _score 与函数值求和的方式可以弱化最终效果， 特别是使用一个较小 factor 因子时：

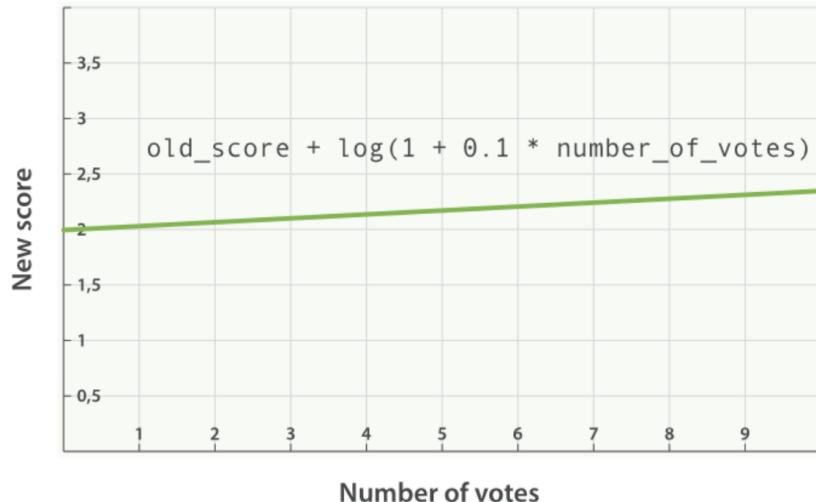
```
GET /blogposts/post/_search
{
  "query": {
    "function_score": {
      "query": {
        "multi_match": {
          "query": "popularity",
          "fields": [ "title", "content" ]
        }
      },
      "field_value_factor": {
        "field": "votes",
        "modifier": "log1p",
        "factor": 0.1
      },
      "boost_mode": "sum" ( 1 )
    }
  }
}
```

将函数计算结果值累加到评分 `_score`。

之前请求的公式现在变成下面这样（参见 Figure 32，“使用 sum 结合受欢迎程度”）：

```
new_score = old_score + log(1 + 0.1 * number_of_votes)
```

Figure 32. 使用 `sum` 结合受欢迎程度



1.4. max_boost

最后，可以使用 `max_boost` 参数限制一个函数的最大效果：

```
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) 无论 field_value_factor 函数的结果如何，最终结果都不会大于 1.5。

注意

max_boost 只对函数的结果进行限制，不会对最终评分 _score 产生直接影响。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-24

- [1. 过滤集提升权重](#)
 - [1.1. 过滤 VS 查询](#)
 - [1.2. 函数functions](#)
 - [1.3. 评分模式score_mode](#)

1. 过滤集提升权重

回到忽略 TF/IDF 里处理过的问题，我们希望根据每个度假屋的特性数量来评分，当时我们希望能用缓存的过滤器来影响评分，现在 function_score 查询正好可以完成这件事情。

到目前为止，我们展现的都是为所有文档应用单个函数的使用方式，现在会用过滤器将结果划分为多个子集（每个特性一个过滤器），并为每个子集使用不同的函数。

在下面例子中，我们会使用 weight 函数，它与 boost 参数类似可以用于任何查询。有一点区别是 weight 没有被 Luence 归一化成难以理解的浮点数，而是直接被应用。

查询的结构需要做相应变更以整合多个函数：

```
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) function_score 查询有个 filter 过滤器而不是 query 查询。
- (2) functions 关键字存储着一个将被应用的函数列表。
- (3) 函数会被应用于和 filter 过滤器（可选的）匹配的文档。
- (4) pool 比其他特性更重要，所以它有更高 weight。
- (5) score_mode 指定各个函数的值进行组合运算的方式。

这个新特性需要注意的地方会在以下小节介绍。

1.1. 过滤 VS 查询

首先要注意的是 filter 过滤器代替了 query 查询，在本例中，我们无须使用全文搜索，只想找到 city 字段中包含 Barcelona 的所有文档，逻辑用过滤比用查询表达更清晰。过滤器返回的所有文档的评分 _score 的值为 1。function_score 查询接受 query 或 filter，如果没有特别指定，则默认使用 match_all 查询。

1.2. 函数functions

functions 关键字保持着一个将要被使用的函数列表。可以为列表里的每个函数都指定一个 filter 过滤器，在这种情况下，函数只会被应用到那些与过滤器匹配的文档，例子中，我们为与过滤器匹配的文档指定权重值 weight 为 1（为与 pool 匹配的文档指定权重值为 2）。

1.3. 评分模式score_mode

每个函数返回一个结果，所以需要一种将多个结果缩减到单个值的方式，然后才能将其与原始评分 _score 合并。评分模式 score_mode 参数正好扮演这样的角色，它接受以下值：

- multiply
函数结果求积（默认）。
- sum
函数结果求和。
- avg
函数结果的平均值。
- max
函数结果的最大值。
- min
函数结果的最小值。
- first
使用首个函数（可以有过滤器，也可能没有）的结果作为最终结果

在本例中，我们将每个过滤器匹配结果的权重 weight 求和，并将其作为最终评分结果，所以会使用 sum 评分模式。

不与任何过滤器匹配的文档会保有其原始评分，_score 值的为 1。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间：2021-07-24

- 1. 随机评分

1. 随机评分

你可能会想知道一致随机评分 (consistently random scoring) 是什么，又为什么会使用它。之前的例子是个很好的应用场景，前例中所有的结果都会返回 1、2、3、4 或 5 这样的最终评分 _score，可能只有少数房子的评分是 5 分，而有大量房子的评分是 2 或 3。

作为网站的所有者，总会希望让广告有更高的展现率。在当前查询下，有相同评分 _score 的文档会每次都以相同次序出现，为了提高展现率，在此引入一些随机性可能会是个好主意，这能保证有相同评分的文档都能有均等相似的展现机率。

我们想让每个用户看到不同的随机次序，但也同时希望如果是同一用户翻页浏览时，结果的相对次序能始终保持一致。

这种行为被称为一致随机 (consistently random)。

random_score 函数会输出一个 0 到 1 之间的数，当种子 seed 值相同时，生成的随机结果是一致的，例如，将用户的会话 ID 作为 seed：

```
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 user's session id" (2)
          }
        }
      ],
      "score_mode": "sum"
    }
  }
}
```

(1) random_score 语句没有任何过滤器 filter，所以会被应用到所有文档。

(2) 将用户的会话 ID 作为种子 seed，让该用户的随机始终保持一致，相同的种子 seed 会产生相同的随机结果。

当然，如果增加了与查询匹配的新文档，无论是否使用一致随机，其结果顺序都会发生变化。

Copyright © Songbai Yang all right reserved, powered by Gitbook该文件修订时间： 2021-07-24

- 1. 越近越好

1. 越近越好

很多变量都可以影响用户对于度假屋的选择，也许用户希望离市中心近点，但如果价格足够便宜，也有可能选择一个更远的住处，也有可能反过来是正确的：愿意为最好的位置付更多的价钱。

如果我们添加过滤器排除所有市中心方圆 1 千米以外的度假屋，或排除所有每晚价格超过 £100 英镑的，我们可能会将用户愿意考虑妥协的那些选择排除在外。

`function_score` 查询会提供一组衰减函数（decay functions），让我们有能力在两个滑动标准，如地点和价格，之间权衡。

有三种衰减函数——`linear`、`exp` 和 `gauss`（线性、指数和高斯函数），它们可以操作数值、时间以及经纬度地理坐标点这样的字段。所有三个函数都能接受以下参数：

- `origin`
中心点或字段可能的最佳值，落在原点 `origin` 上的文档评分 `_score` 为满分 1.0。
- `scale`
衰减率，即一个文档从原点 `origin` 下落时，评分 `_score` 改变的速度。（例如，每 £10 欧元或每 100 米）。
- `decay`
从原点 `origin` 衰减到 `scale` 所得的评分 `_score`，默认值为 0.5。
- `offset`
以原点 `origin` 为中心点，为其设置一个非零的偏移量 `offset` 覆盖一个范围，而不只是单个原点。在范围 $-\text{offset} \leq \text{origin} \leq +\text{offset}$ 内的所有评分 `_score` 都是 1.0。

这三个函数的唯一区别就是它们衰减曲线的形状，用图来说明会更为直观（参见 Figure 33，“衰减函数曲线”）。

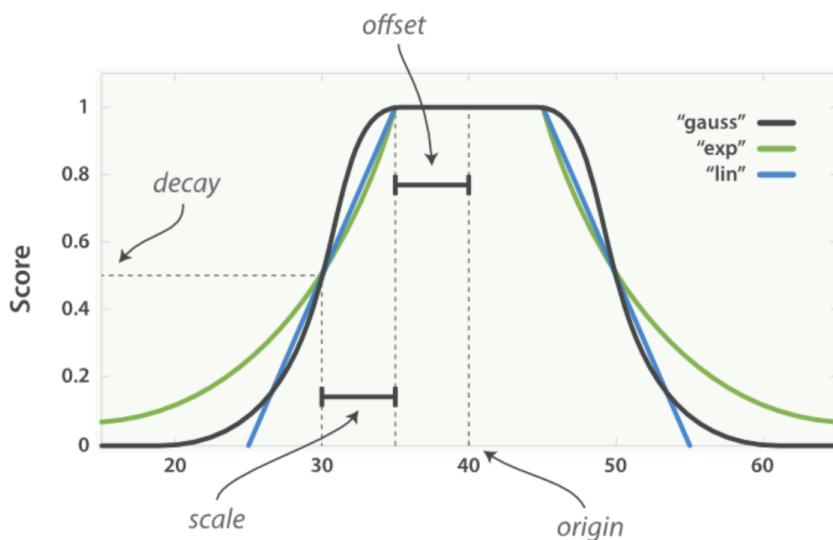


图 Figure 33, “衰减函数曲线” 中所有曲线的原点 origin (即中心点) 的值都是 40 , offset 是 5 , 也就是在范围 $40 - 5 \leq value \leq 40 + 5$ 内的所有值都会被当作原点 origin 处理——所有这些点的评分都是满分 1.0 。

在此范围之外, 评分开始衰减, 衰减率由 scale 值 (此例中的值为 5) 和 衰减值 decay (此例中为默认值 0.5) 共同决定。结果是所有三个曲线在 $origin \pm (offset + scale)$ 处的评分都是 0.5 , 即点 30 和 50 处。

linear 、 exp 和 gauss (线性、指数和高斯) 函数三者之间的区别在于范围 ($origin \pm (offset + scale)$) 之外的曲线形状:

- linear 线性函数是条直线, 一旦直线与横轴 0 相交, 所有其他值的评分都是 0.0 。
- exp 指数函数是先剧烈衰减然后变缓。
- gauss 高斯函数是钟形的——它的衰减速率是先缓慢, 然后变快, 最后又放缓。

选择曲线的依据完全由期望评分 _score 的衰减速率来决定, 即距原点 origin 的值。

回到我们的例子: 用户希望租一个离伦敦市中心近 ({ "lat": 51.50, "lon": 0.12}) 且每晚不超过 £100 英镑的度假屋, 而且与距离相比, 我们的用户对价格更为敏感, 这样查询可以写成:

```
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) location 字段以地理坐标点 geo_point 映射。 (2) price 字段是数值。

(3) 参见 理解价格语句 , 理解 origin 为什么是 50 而不是 100 。

(4) price 语句是 location 语句权重的两倍。

location 语句可以简单理解为 :

- 以伦敦市中作为原点 origin 。
- 所有距原点 origin 2km 范围内的位置的评分是 1.0 。
- 距中心 5km （ offset + scale ）的位置的评分是 0.5 。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间： 2021-07-24

- 1. 理解 price 价格语句

1. 理解 price 价格语句

price 语句使用了一个小技巧：用户希望选择 £100 英镑以下的度假屋，但是例子中的原点被设置成 £50 英镑，价格不能为负，但肯定是越低越好，所以 £0 到 £100 英镑内的所有价格都认为是比较好的。

如果我们将原点 origin 被设置成 £100 英镑，那么低于 £100 英镑的度假屋的评分会变低，与其这样不如将原点 origin 和偏移量 offset 同时设置成 £50 英镑，这样就能使只有在价格高于 £100 英镑（origin + offset）时评分才会变低。

Tip

weight 参数可以被用来调整每个语句的贡献度，权重 weight 的默认值是 1.0。这个值会先与每个句子的评分相乘，然后再通过 score_mode 的设置方式合并。

Copyright © Songbai Yang all right reserved, powered by Gitbook
该文件修订时间：2021-07-25

- 1. 脚本评分

1. 脚本评分

最后，如果所有 function_score 内置的函数都无法满足应用场景，可以使用 script_score 函数自行实现逻辑。举个例子，想将利润空间作为因子加入到相关度评分计算，在业务中，利润空间和以下三点相关：

- price 度假屋每晚的价格。
- 会员用户的级别——某些等级的用户可以在每晚房价高于某个 threshold 阈值价格的时候享受折扣 discount。
- 用户享受折扣后，经过议价的每晚房价的利润 margin。

计算每个度假屋利润的算法如下：

```
if (price < threshold) {
    profit = price * margin
} else {
    profit = price * (1 - discount) * margin;
}
```

我们很可能不想用绝对利润作为评分，这会弱化其他如地点、受欢迎度和特性等因子的作用，而是将利润用目标利润 target 的百分比来表示，高于目标的利润空间会有一个正向评分（大于 1.0），低于目标的利润空间会有一个负向分数（小于 1.0）：

```
if (price < threshold) {
    profit = price * margin
} else {
    profit = price * (1 - discount) * margin
}
return profit / target
```

Elasticsearch 里使用 [Groovy](#) 作为默认的脚本语言，它与 JavaScript 很像，上面这个算法用 Groovy 脚本表示如下：

```
``` price = doc['price'].value (1) margin = doc['margin'].value (1)
```

```
if (price < threshold) { (2) return price margin / target } return price (1 - discount) *
margin / target (2)
```

(1) price 和 margin 变量可以分别从文档的 price 和 margin 字段提取。

(2) threshold、discount 和 target 是作为参数 params 传入的。

最终我们将 script\_score 函数与其他函数一起使用：

```
GET /_search { "function_score": { "functions": [{ ...location clause... }, (1){
...price clause... }, (1)
{ "script_score": { "params": { (2) "threshold": 80, "discount": 0.1, "target": 10 },
"script": "price = doc['price'].value; margin = doc['margin'].value; if (price <
threshold) { return price margin / target }; return price (1 - discount) * margin /
target;" (3)} }] } } ```

(1) location 和 price 语句在 衰减函数 中解释过。
```

- (2) 将这些变量作为参数 `params` 传递，我们可以查询时动态改变脚本无须重新编译。
- (3) JSON 不能接受内嵌的换行符，脚本中的换行符可以用 `\n` 或 ; 符号替代。

这个查询根据用户对地点和价格的需求，返回用户最满意的文档，同时也考虑到我们对于盈利的要求。

TIP

`script_score` 函数提供了巨大的灵活性，可以通过脚本访问文档里的所有字段、当前评分 `_score` 甚至词频、逆向文档频率和字段长度规范值这样的信息（参见 see [脚本对文本评分](#)）。

有人说使用脚本对性能会有影响，如果确实发现脚本执行较慢，可以有以下三种选择：(1) 尽可能多的提前计算各种信息并将结果存入每个文档中。

(2) Groovy 很快，但没 Java 快。可以将脚本用原生的 Java 脚本重新实现。（参见 [原生 Java 脚本](#)）。

(3) 仅对那些最佳评分的文档应用脚本，使用 [重新评分](#) 中提到的 `rescore` 功能。

Copyright © Songbai Yang all right reserved, powered by Gitbook  
该文件修订时间：2021-07-25

- 1. 可插拔的相似度算法
  - 1.1. Okapi BM25
  - 1.2. 词频饱和度
  - 1.3. 字段长归一化
  - 1.4. BM25

## 1. 可插拔的相似度算法

在进一步讨论相关度和评分之前，我们会以一个更高级的话题结束本章节的内容：  
可插拔的相似度算法（Pluggable Similarity Algorithms）。Elasticsearch 将 [实用评分算法](#) 作为默认相似度算法，它也能够支持其他的一些算法，这些算法可以参考 [相似度模块](#) 文档。

### 1.1. Okapi BM25

能与 TF/IDF 和向量空间模型媲美的就是 Okapi BM25，它被认为是当今最先进的排序函数。BM25 源自 [概率相关模型](#)（probabilistic relevance model），而不是向量空间模型，但这个算法也和 Lucene 的实用评分函数有很多共通之处。

BM25 同样使用词频、逆向文档频率以及字段长归一化，但是每个因子的定义都有细微区别。与其详细解释 BM25 公式，倒不如将关注点放在 BM25 所能带来的实际好处上。

### 1.2. 词频饱和度

TF/IDF 和 BM25 同样使用 [逆向文档频率](#) 来区分普通词（不重要）和非普通词（重要），同样认为（参见 [词频](#)）文档里的某个词出现次数越频繁，文档与这个词就越相关。

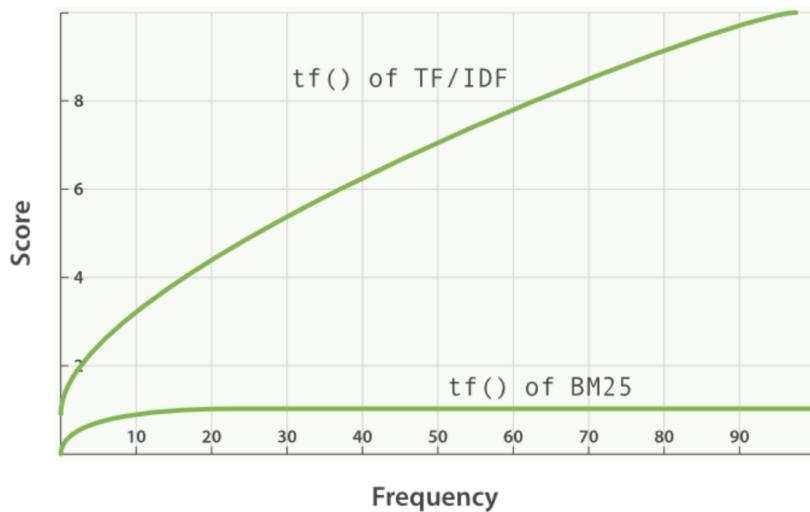
不幸的是，普通词随处可见，实际上一个普通词在同一个文档中大量出现的作用会由于该词在所有文档中的大量出现而被抵消掉。

曾经有个时期，将最普通的词（或停用词，参见 [停用词](#)）从索引中移除被认为是一种标准实践，TF/IDF 正是在这种背景下诞生的。TF/IDF 没有考虑词频上限的问题，因为高频停用词已经被移除了。

Elasticsearch 的 standard 标准分析器（string 字段默认使用）不会移除停用词，因为尽管这些词的重要性很低，但也不是毫无用处。这导致：在一个相当长的文档中，像 the 和 and 这样词出现的数量会高得离谱，以致它们的权重被人为放大。

另一方面，BM25 有一个上限，文档里出现 5 到 10 次的词会比那些只出现一两次的对相关度有着显著影响。但是如图 TF/IDF 与 BM25 的词频饱和度所见，文档中出现 20 次的词几乎与那些出现上千次的词有着相同的影响。

这就是 [非线性词频饱和度](#)（nonlinear term-frequency saturation）。

**Figure 34. TF/IDF 与 BM25 的词频饱和度**

### 1.3. 字段长归一化

在 [字段长归一化](#) 中，我们提到过 Lucene 会认为较短字段比较长字段更重要：字段某个词的频度所带来的的重要性会被这个字段长度抵消，但是实际的评分函数会将所有字段以同等方式对待。它认为所有较短的 title 字段比所有较长的 body 字段更重要。

BM25 当然也认为较短字段应该有更多的权重，但是它会分别考虑每个字段内容的平均长度，这样就能区分短 title 字段和长 title 字段。

Tip

在 [查询时权重提升](#) 中，已经说过 title 字段因为其长度比 body 字段自然有更高的权重提升值。由于字段长度的差异只能应用于单字段，这种自然的权重提升会在使用 BM25 时消失。

### 1.4. BM25

不像 TF/IDF，BM25 有一个比较好的特性就是它提供了两个可调参数：

- k1  
这个参数控制着词频结果在词频饱和度中的上升速度。默认值为 1.2。值越小饱和度变化越快，值越大饱和度变化越慢。
- b  
这个参数控制着字段长归一值所起的作用，0.0 会禁用归一化，1.0 会启用完全归一化。默认值为 0.75。

在实践中，调试 BM25 是另外一回事，k1 和 b 的默认值适用于绝大多数文档集合，但最优值还是会因为文档集不同而有所区别，为了找到文档集合的最优值，就必须对参数进行反复修改验证。

- [1. 更改相似度](#)
- [2. 配置BM25](#)

## 1. 更改相似度

相似度算法可以按字段指定，只需在映射中为不同字段选定即可：

```
PUT /my_index
{
 "mappings": {
 "doc": {
 "properties": {
 "title": {
 "type": "string",
 "similarity": "BM25"
 },
 "body": {
 "type": "string",
 "similarity": "default"
 }
 }
 }
}
```

- (1) title 字段使用 BM25 相似度算法。  
(2) body 字段用默认相似度算法（参见 实用评分函数）。
- 目前，Elasticsearch 不支持更改已有字段的相似度算法 similarity 映射，只能通过为数据重新建立索引来达到目的。

## 2. 配置BM25

配置相似度算法和配置分析器很相似，自定义相似度算法可以在创建索引时指定，例如：

```
PUT /my_index
{
 "settings": {
 "similarity": {
 "my_bm25": {
 "type": "BM25",
 "b": 0
 }
 }
 },
 "mappings": {
 "doc": {
 "properties": {
 "title": {
 "type": "string",
 "similarity": "my_bm25" (3)
 },
 "body": {
 "type": "string",
 "similarity": "BM25" (4)
 }
 }
 }
 }
}
```

- (1) 创建一个基于内置 BM25，名为 my\_bm25 的自定义相似度算法。
- (2) 禁用字段长度规范化（field-length normalization）。参见 [调试 BM25](#)。
- (3) title 字段使用自定义相似度算法 my\_bm25 。
- (4) 字段 body 使用内置相似度算法 BM25 。

Tip

定义的相似度算法可以通过关闭索引，更新索引设置，开启索引这个过程进行更新。这样可以无须重建索引又能试验不同的相似度算法配置。

Copyright © Songbai Yang all right reserved, powered by Gitbook  
该文件修订时间： 2021-07-25

- 1. 调试相关度是最后 10% 要做的事情

## 1. 调试相关度是最后 10% 要做的事情

本章介绍了 Lucene 是如何基于 TF/IDF 生成评分的。理解评分过程是非常重要的，这样就可以根据具体的业务对评分结果进行调试、调节、减弱和定制。

实践中，简单的查询组合就能提供很好的搜索结果，但是为了获得具有成效的搜索结果，就必须反复推敲修改前面介绍的这些调试方法。

通常，经过对策略字段应用权重提升，或通过对查询语句结构的调整来强调某个句子的重要性这些方法，就足以获得良好的结果。有时，如果 Lucene 基于词的 TF/IDF 模型不再满足评分需求（例如希望基于时间或距离来评分），则需要更具侵略性的调整。

除此之外，相关度的调试就有如兔子洞，一旦跳进去就很难再出来。最相关这个概念是一个难以触及的模糊目标，通常不同人对文档排序又有着不同的想法，这很容易使人陷入持续反复调整而没有明显进展的怪圈。

我们强烈建议不要陷入这种怪圈，而要监控测量搜索结果。监控用户点击最顶端结果的频次，这可以是前 10 个文档，也可以是第一页的；用户不查看首次搜索的结果而直接执行第二次查询的频次；用户来回点击并查看搜索结果的频次，等等诸如此类的信息。

这些都是用来评价搜索结果与用户之间相关程度的指标。如果查询能返回高相关的文档，用户会选择前五中的一个，得到想要的结果，然后离开。不相关的结果会让用户来回点击并尝试新的搜索条件。

一旦有了这些监控手段，想要调试查询就并不复杂，稍作调整，监控用户的行为改变并做适当反复尝试。本章介绍的一些工具就只是工具而已，要想物尽其用并将搜索结果提高到极高的水平，唯一途径就是需要具备能评价度量用户行为的强大能力。

Copyright © Songbai Yang all right reserved, powered by Gitbook  
该文件修订时间：2021-07-25