

初识elasticsearch

Robin Fu

微信 robinfu188

2020年07月

序

- 为什么要做这次分享
 - 我们经常在一知半解的状态下使用某个工具
 - 听起来很炫酷，还可以写进简历
 - 但是偶尔会发生一些事情：
 - 突然又有哪个线程堵住了；
 - 分布式锁好像没锁住；
 - akka集群节点传染性自杀；
 - geode发个版莫名其妙挂了...
 - 希望能激发大家对es以及其他常用工具的兴趣
- 分享有什么好处
 - 传播知识，这很显然
 - 巩固知识，准备的过程中阅读大量的资料，准备各种案例，做大量的实验



跳过教程直接进入游戏的新玩家

es简介

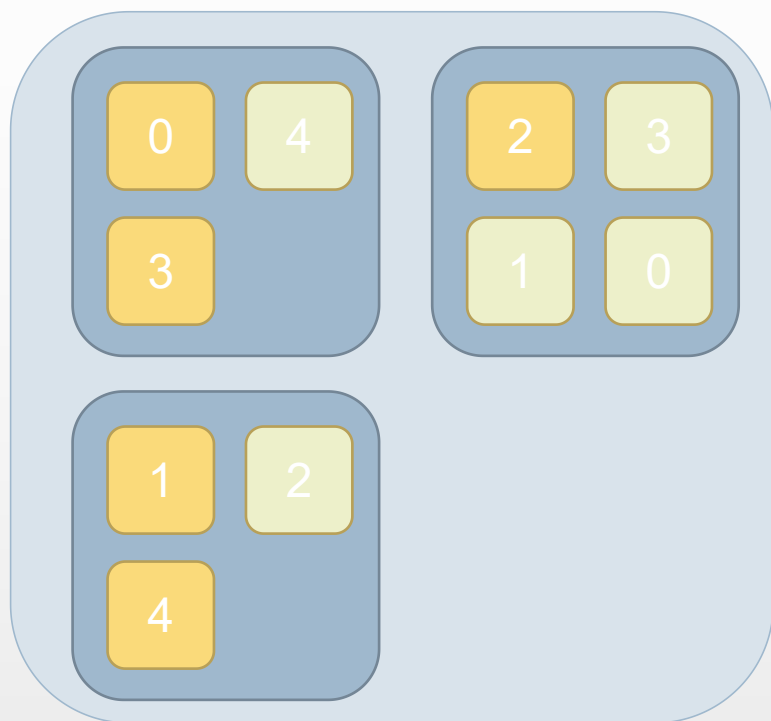
- 基于Apache Lucene的开源分布式搜索引擎
- 应用场景
 - 日志记录和分析（接口调用情况）
 - 采集和组合公共数据（用户行为、点击事件）
 - 全文检索（酒店搜索）
 - 数据可视化（借助kibana制作各种图表）
- 我们应该了解它才能用得更好
- 自己在电脑上装一个es玩玩
 - 在Linux子系统中，下载，解压，启动，欧了
 - 我们用的是6.8.2版本（并不新
 - 此处有案例讲解

一、理解逻辑和物理设计

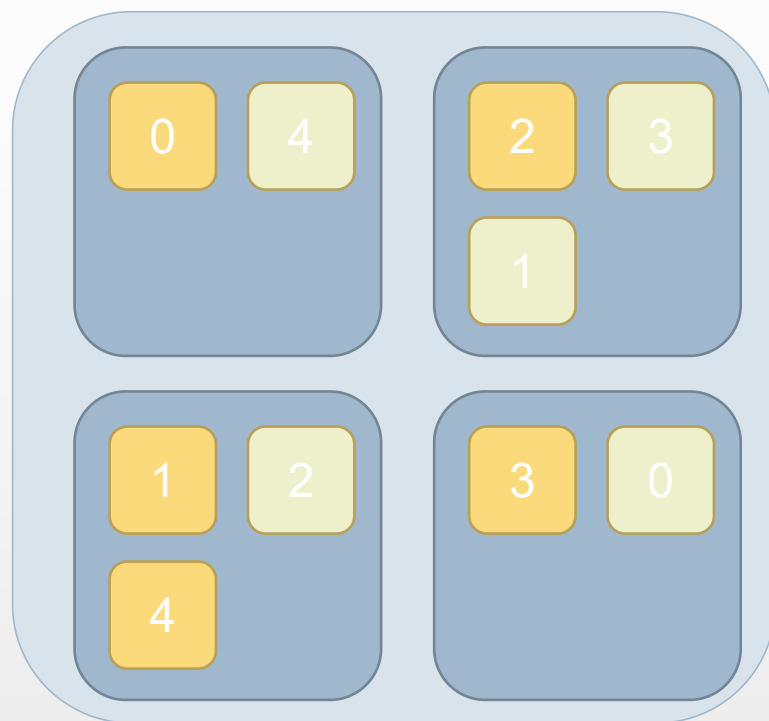
关键词介绍

- 集群(cluster)
 - es是分布式的，由多个节点构成集群，集群具有很强的扩展性
- 节点(node)
 - 每个启动的es实例就是一个节点
 - 可以随时加入和脱离集群
- 索引(index)
 - 名词：一堆字段相似的文档的集合，类似数据库的一张表
 - 动词：将文档写入某个索引
- 文档(document)
 - 一条es的记录
- 分片(shard)
 - es处理的最小单元
 - 一个分片是一个lucene索引
 - 一个包含倒排索引的文件目录
 - 分片越多搜索越慢
- 分段(segment)
 - lucene索引再分割成小单元
 - 分段越多搜索越慢
 - 分段不会被修改
 - 索引新的文档会创建新的分段
 - 分段会持续地被合并（像小水滴变成大水滴）
 - 删除文档的时候不会真的删除（只是标记）

分片：扩展和容灾



es集群



加入一个新节点



主分片



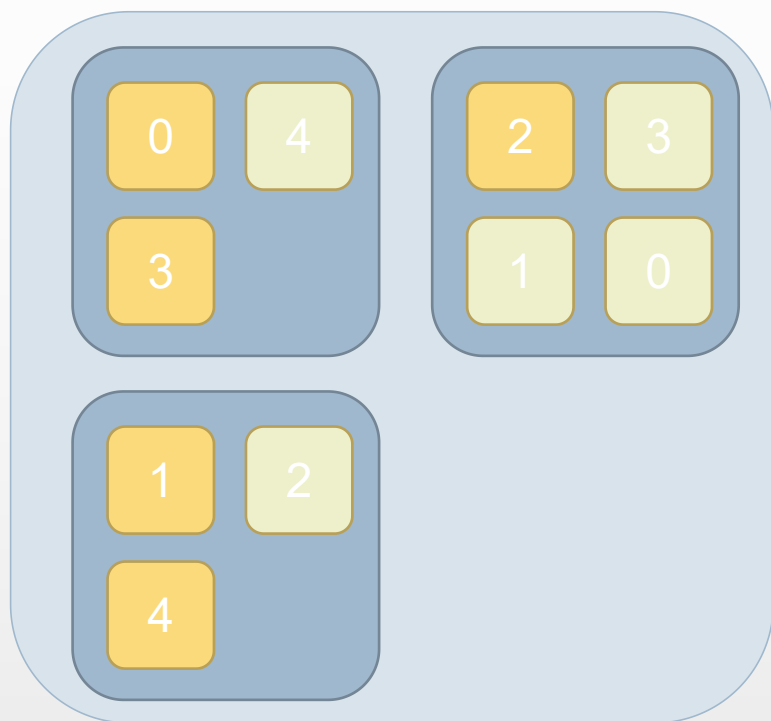
副本分片



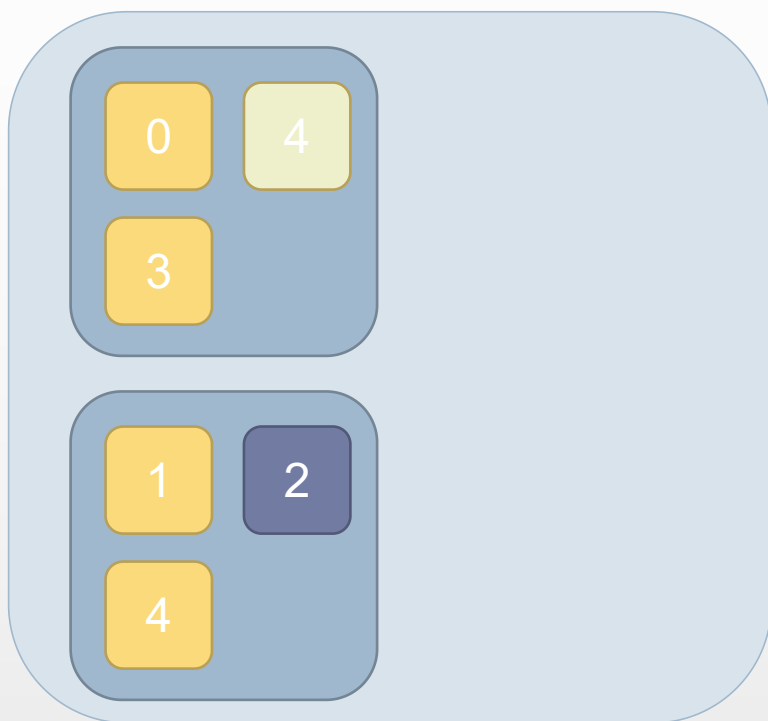
节点

- 一个索引的所有分片会自动均匀分布在所有节点中。
- 加入新节点后，原集群节点的分片，会部分迁移到新节点。
- 设置分片数量稍微大于节点数量，有利于横向扩容时，分片蔓延到所有新节点（每个节点都有分片是最理想状态）。
- 主分片和所有副本分片都就绪时，索引的健康状态是绿色。

分片：扩展和容灾



es集群



挂掉一个节点



主分片



副本分片

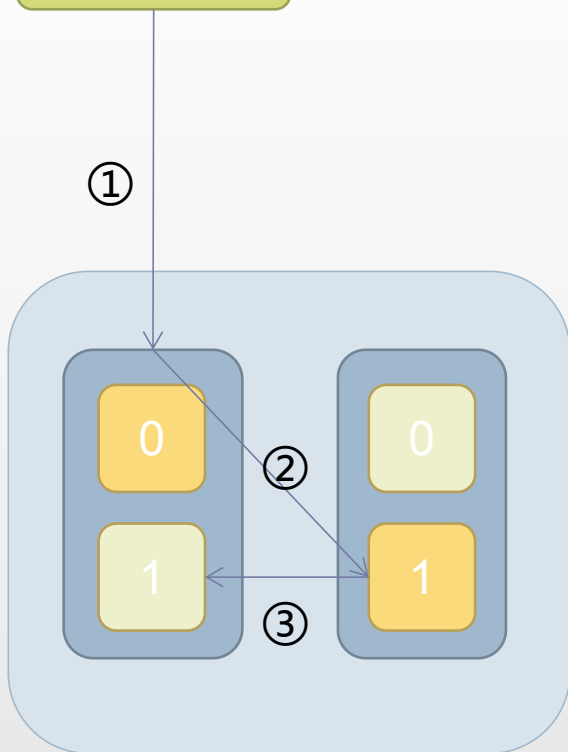


节点

- 挂掉了 n 个节点时，如果副本分片是 n ，那么剩下的副本分片将自动提升为主分片。
- 然后所有的主分片能构成完整的索引，但是副本分片缺失，此时索引健康状态是黄色。
- 如果挂掉 $n+1$ 个节点，主分片将缺失，健康状态是红色。
- 根据实际情况设置副本数量（副本太多会影响性能）
- 通常同时挂掉两个的概率不高，1个副本可以满足常规容灾要求。

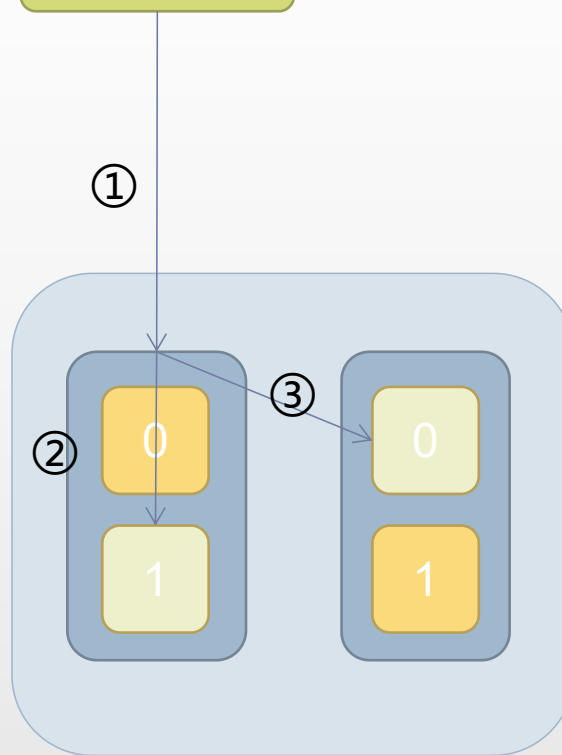
索引和搜索数据

索引请求



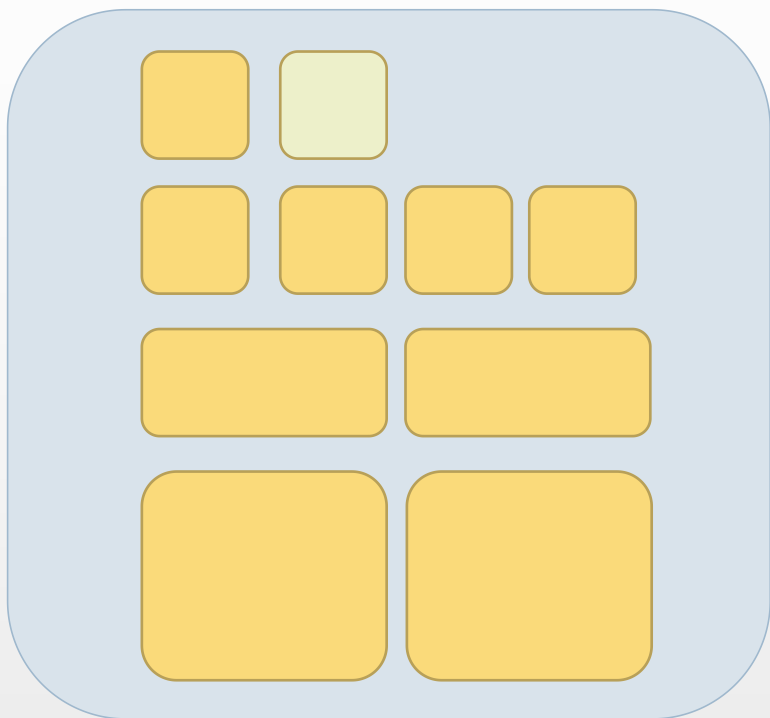
- 索引文档请求到一个节点
- 文档被随机到一个主分片
- 从主分片同步到副本分片
- 返回成功的结果
- 副本分片越多，索引数据越慢，因为要所有副本都完成才算完成

搜索请求

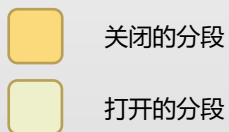


- 搜索请求到一个节点
- 节点转发请求到本节点的一个分片，到其他节点的另一个分片
- 所有分片都返回搜索结果到发起节点
- 发起节点返回搜索结果到请求方
- 不同节点上的主分片+副本分片的总数越多，请求被分摊得越多，并发搜索性能越好
- 但如果节点数很少，分片都集中到少数节点上，搜索速度会变慢，因为增加了开销，实际没有分摊负载
- 单个搜索无法通过分片加速

分段



分片



- 写入
 - 只能写打开的分段（为了避免冲突合并）
 - 删除是假删除，也是往打开的分段写
 - 分段大小超过一定阈值，会触发分段合并
 - 小分段合并成大分段，为了查询加速，但合并过程是先创建一个大分段，把俩小的放进去，再删掉俩小的。这个过程会**耗费大量资源**。
 - 如果希望写入快，应该**避免频繁分段合并**。
- 读取
 - 只能读关闭的分段（所以叫**准实时**）
 - 对一个分片查询，会等它所有的分段结果，所以分段过多，
 - 刷新时，会关闭一批分段，这时候数据才能被查到
 - 刷新频率太快会导致**分段碎片多**
 - 刷新频率慢会导致读写**实时性低**
- 一切都是为了更快
 - 弱化关系、弱化一致性，都是为了速度
 - 想读得快，就要牺牲写速度，反之亦然

索引分片不是免费的吗？

- 每个索引和分片都会产生一定的资源开销
- 每个索引，映射和状态的相关信息都存储在集群状态中
 - 存储在内存中，以便快速访问
 - 分片数量过多，会导致集群状态过大
 - 这会导致更新变慢，因为所有更新都需要通过单线程完成，从而在将变更分发到整个集群之前确保一致性
- 分片有一部分数据需要保存在内存中
 - 这部分数据也会占用堆内存空间
 - 这包括存储分片级别以及段级别信息的数据结构
 - 因为只有这样才能确定数据在磁盘上的存储位置
- 在单个节点上存储尽可能多的数据
 - 管理堆内存使用量
 - 尽可能减少开销
 - 节点的堆内存空间越多，其能处理的数据和分片就越多

官网建议

- Jvm heap每1G不超过20个分片
- 每个分片大小在20G-40G
- <https://www.elastic.co/cn/blog/how-many-shards-should-i-have-in-my-elasticsearch-cluster>

二、索引、更新和删除数据

字段类型映射 (mapping)

- 类似数据库的字段类型定义，你定义了一个字段，就要指定一个类型
- 如果不指定字段类型，es在插入第一条数据时，会自动创建一个index，同时帮你创建映射
- 字段如果和定义的类型不匹配，会插入失败
 - 因此同一个字段不能有时候是字符串，有时候是数组
 - 最好提前定义映射而不是依赖自动
- 字段的mapping配置中，除了指定类型，还可以为字段指定参数
 - analyzer：指定分析器
 - boost：指定字段的分值加成
- 通过模板 (template) 创建索引
 - 除了每次创建index前，手动指定index的mapping和配置（类似数据库建表操作）
 - 新建一个index的时候，可以自动从模板里获取index的mapping以及其他的设置（分片、副本数等）
 - 这是非常常用的操作

字段类型

- 字段
 - 核心类型
 - **text和keyword，都是字符串，keyword不被分析**
 - 数字类型：long, integer, short, byte, double, float, half_float, scaled_float
 - 日期类型：date
 - 布尔类型：boolean
 - 二进制类型：binary
 - 范围类型：integer_range, float_range, long_range, double_range, date_range
 - 复合类型
 - **对象类型：通常的一个json会被扁平化**
 - **嵌套类型：json字段之间的关系会被保留**
 - 地理类型
 - 地理点和地理区域
 - 特殊类型
 - ip, completion, token_count, murmur3, percolator, join
- 数组
 - 单个字段的列表形式
 - 数组只要一个元素命中即整条doc命中
- 多元字段
 - 内置的一些特殊字段，以下划线开头
 - `_index, _id, _type, _uid`
 - `_source, _size, _all, _field_names`
 - **`_routing`：指定文档在哪个分片**
 - `_meta`
- 参考官网：
<https://www.elastic.co/guide/en/elasticsearch/reference/6.8/mapping-types.html>

动态设置mapping

- 如果事先没有创建index，或者是索引了新字段，字段mapping将会被动态设置
- Dynamic field mapping
 - Es会根据json字段自动判断类型，甚至能够发现string里面填的是日期、数值还是文本来设置字段类型
 - <https://www.elastic.co/guide/en/elasticsearch/reference/6.8/dynamic-field-mapping.html>
- Dynamic templates
 - 可以配置映射模板，自定义类型识别
 - <https://www.elastic.co/guide/en/elasticsearch/reference/6.8/dynamic-templates.html>
- Index templates
 - 索引模板，通过索引名命中模板，模板中设置字段类型
 - <https://www.elastic.co/guide/en/elasticsearch/reference/6.8/indices-templates.html>
- 示例
 - GET index名称/_mapping
 - GET _template
- 注意
 - 字段是否分析：不分析就不能全文检索这个字段，省性能
 - 字段是否索引：不索引就不能搜这个字段，省性能

分析字段

- 这是es的关键概念，先说个大概，后续会深入了解
- 定义：解析、转变、分解文本，使得搜索更加相关
- 分析包括三个步骤
 - **字符过滤（过滤器）**：使用字符过滤器转变字符（比如：大写变小写）
 - **文本切分为分词（分词器）**：将文本切分为单个或者多个分词（比如：英文文本用空格切为一堆单词）
 - **分词过滤（分词过滤器）**：转变每个分词（比如：把a an of 这类词干掉，或者复数单词转为单数单词）

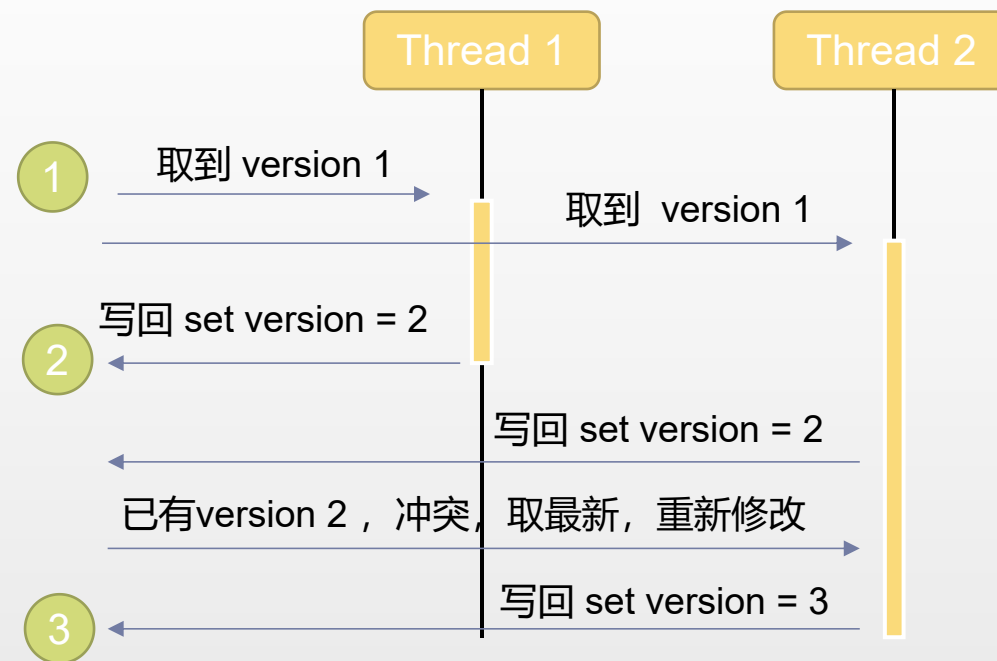


更新文档

- 前面提过segment创建之后**不能修改**
- 因此文档更新实际上是建了一条新的然后删掉旧的
- 其实删除也**不是真删**，而是加上一条删除标记
 - 分段合并的时候会真删
 - 删除会消耗性能
 - 整个index删除是最快的
- 文档更新包括：检索文档（按id）、处理文档、重新索引文档
- 有三种方式：整条更新、**字段合并**、若存在则放弃更新
- 可以使用自动生成id来插入新文档，这样可以节省检索文档所耗费的资源，加快索引速度

通过文档版本实现并发控制

- es显然是分布式的，那么就会有并发问题
- 在一个更新获取原文档进行修改期间，可能有另一个更新也在修改这篇文档，那么第一个更新就丢了
- es用文档版本号来解决这个问题（类似乐观锁）
 - 为每个文档设置一个版本号
 - 文档创建时版本号是1
 - 当更新后，版本号变成2
 - 如果此时有另一个更新，版本号也将是2，此更新结束时发现已经有了一个2，那么将产生冲突
 - 发生冲突后，重试这个更新操作，如果不再冲突，那么完成更新，版本号设置为3
- 可以更精确地控制冲突
 - 默认情况下遇到冲突会更新失败，通过参数`retry_on_conflict`，控制重试次数，默认为0（不重试）
 - 可以显式指定版本号（插入和更新都可以），而不是默认取最新版本
 - 可以使用外部版本号，比如时间戳



删除文档

- 删除文档
 - 可以通过id删除单个，也可以通过条件批量删除
 - 类似`delete .. from .. where ..`
 - 删除文档拖慢查询和进一步的索引
 - 删除只是标记为删除
 - 搜索的时候还要检查一遍命中的文档是否已经被删掉
 - 分段合并的时候才彻底删除
 - 不推荐使用
- 删除索引
 - 删除索引是很快的，因为是直接移除索引相关的分片文件
 - 删除是不可恢复的，在生产环境上也没有权限控制，一定要小心操作（7.0+可控制权限）
 - `DELETE my_index`
 - 小心！！ **DELETE _all**会直接瞬间清空所有索引！！

删除文档

- 关闭索引

- 除了删除，还有一个更安全操作，就是关闭索引
- 索引关闭之后，不能读取和写入，直到再次打开
- 关闭后的索引只占磁盘，非常cheap，因此我们通常会关闭而不是删除索引

```
POST /my_index/_close
POST /my_index/_open
```

- <https://www.elastic.co/guide/en/elasticsearch/reference/6.8/indices-open-close.html>

- 冻结索引

- 扩展包功能
- 介于打开和关闭之间
 - 不能写入
 - 分片开销很小

```
POST /my_index/_freeze
POST /my_index/_unfreeze
```

```
GET /my_index/_search?ignore_throttled=false
```

- 腾讯云版es做了限制，需要特殊参数才能搜

- <https://www.elastic.co/guide/en/elasticsearch/reference/6.8/freeze-index-api.html>

reindex

- 复制一个索引
 - 可用于重建索引
 - 可用于提取字段
 - 可以跨集群复制
- 改变索引配置
 - 分段一旦生成就不能修改，因此索引一旦创建就无法改变
 - 有些索引的配置也是不可改变的，比如分片数量、mapping映射等
 - 只能通过重建索引修改
- 提取字段
 - 有时字段需要通过脚本处理后才能满足新的使用需求，
 - 比如只存了航班AA571，没有单独存航司，需要按航司聚类
 - 可以用脚本字段聚类，但不建议使用
 - 可以通过reindex，写脚本来索引新字段。
- 注意，reindex操作不会复制索引的配置，需要提前设置，或者配置template
- reindex之前最好先把目标配置的副本数减为0，并关闭刷新，加快写入
- <https://www.elastic.co/guide/en/elasticsearch/reference/6.8/docs-reindex.html>

```
POST _reindex
{
  "source": {
    "index": "twitter"
  },
  "dest": {
    "index": "new_twitter"
  }
}
```

自动管理索引生命周期

- ILM (index lifecycle management (ILM) APIs)
 - 自动管理生命周期
 - x-pack的功能，腾讯云版本有支持
 - <https://www.elastic.co/guide/en/elasticsearch/reference/6.8/index-lifecycle-management.html>
- rollover滚动存储
 - 可让分片大小均匀在30-40G
- shrink缩减分片
 - 写的时候分片多可加速
 - 读的时候收缩分片减小内存消耗
- allocate分片节点感知
 - 冷热分离
 - 远期日志放到冷节点
 - 省钱 + 延长日志存放时间
- forcemerge压缩分段

```
{
  "policy": {
    "phases": {
      "hot": {
        "actions": {
          "rollover": {
            "max_size": "50GB",
            "max_age": "1d"
          }
        }
      },
      "warm": {
        "min_age": "1d",
        "actions": {
          "readonly": {},
          "forcemerge": {
            "max_num_segments": 1
          },
          "shrink": {
            "number_of_shards": 1
          },
          "allocate": {
            "include": {
              "box_type": "hot,warm"
            }
          }
        }
      },
      "cold": {
        "min_age": "3d",
        "actions": {
          "freeze": {},
          "allocate": {
            "include": {
              "box_type": "cold"
            }
          }
        }
      },
      "delete": {
        "min_age": "1d",
        "actions": {
          "delete": {}
        }
      }
    }
  }
}
```

三、搜索数据

搜索上下文

- 分为查询上下文（query context）和过滤上下文（filter context）
 - 区别在于过滤器不计算相关性，只关心是否命中条件
 - 计算相关性需要计算匹配度分值，耗费性能
 - 匹配度分值都是实时计算，无法缓存
 - 应该尽量使用过滤查询以减少性能消耗加快查询速度
- kibana的搜索框和filter是过滤上下文
- <https://www.elastic.co/guide/en/elasticsearch/reference/6.8/query-filter-context.html>

42 hits

resp-msg:*character*

_type: "verify-price" origin-supplier: "API-QUNARAPI" NOT resp: "200" Add a filter +

flight-order-*

分类

- full-text-queries
 - 全文检索，被查询的字段需要被分析
 - 查询条件也会被分析
- term-level-queries
 - 精确匹配查询，查询条件不会被分析
 - 通常用于结构化的数据比如数字、日期、枚举、keyword
 - 对于被分析过的字段也可以用
 - 有一个类似于分析的功能normalizer，和analyzer类似，只是不分词
- compound-queries
 - 嵌套查询，可以嵌套基本查询或者是另一个嵌套
 - 比如与、或、非等等
- joining-queries
 - 类似关系型数据库那样的关联查询
- geo-queries
 - 地理信息查询
- specialized-queries
 - 特殊查询，比如脚本
- span queries
 - 跨度查询，将分词之间的距离纳入查询
 - 被查询的字段需要被分析
- kibana的搜索可以使用本页所有的查询
- 参考文档：
<https://www.elastic.co/guide/en/elasticsearch/reference/6.8/query-dsl.html>

full-text-queries

- **match** query
 - 最基本的全文检索查询，支持单词查询、模糊匹配、短语查询、近义词查询
- **match_phrase** query
 - 类似match，专门查询短语，可以指定短语的间隔slop(默认是0)
 - 比如is test 可以命中this is a test
- **match_phrase_prefix** query
 - 类似短语查询，但最后一个单词是前缀查询，用于最后一个单词想不起来的情况
 - 比如is t可以命中this is a test
- **multi_match** query
 - 把match查询用在多个字段上
- **common_terms** query
 - 给非普通单词更大的权重
 - 比如eat是普通单词，robinfu是特殊单词
- **query_string** query
 - 使用Lucene查询语法的查询，
 - 可以指定各种AND|OR|NOT查询条件，
 - 而且支持在一条语句里对多字段查询
 - kibana的查询框就是用这个，
 - es文档说仅适用于高级玩家
- **simple_query_string**
 - 傻瓜版的query_string，
 - 可以兼容错误的语法，不会搞挂查询，
 - 适合当作搜索框直接暴露给用户

term-level-queries

- **term query**
 - 精确匹配整个查询语句
- **terms query**
 - 类似term, 可以传入一个数组, 匹配一个即可
- **terms_set query**
 - 类似terms, 可以指定匹配条件数,
 - 支持脚本通过计算指定
- **range query**
 - 范围查询,
 - 可以按区间查日期、数字、甚至字符串
- **exists query**
 - 非空查询
- **prefix query**
 - 前缀匹配
- **wildcard query**
 - 通配符查询, 支持单个? 和多个*
 - 通配符放在越前面, 查询效率越低
 - 因此es禁止其放在最前面
- **regexp query**
 - 正则表达式查询
 - 使用不当会造成效率低下的查询
 - 不要出现过度通配
- **fuzzy query**
 - 模糊查询, 比如ab可以命中ba
- **type query**
 - 类型查询, 指定被查询字段的mapping类型
- **ids query**
 - Id查询, 可指定多个id

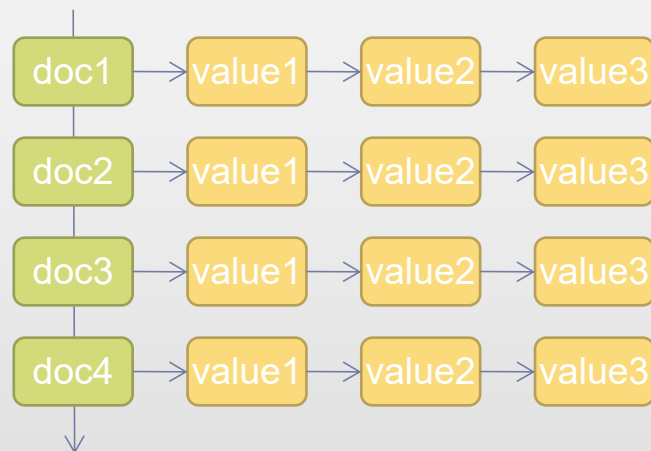
Compound queries

- `constant_score` query
 - 包裹住的查询，会使用filter上下文，不计算相关性得分，可以指定常量分值
- `bool` query
 - 最常用的组合查询，与、或、非、filter，可嵌套
- `dis_max` query
 - 对多个子查询的得分取最高
 - 如果有子查询得分相近，还有加成选项
- `function_score` query
 - 可以对子查询的得分进行复杂计算，比如最大、最小、平均、随机、各种复杂的数学运算
- `boosting` query
 - 可以对子查询进行加分positive或者减分negative
 - 区别于bool query里的NOT，不是去掉，而是降低命中者的权重

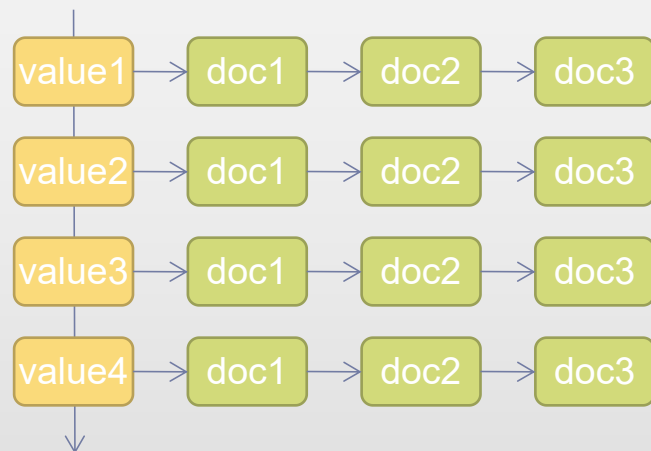
四、分析数据

什么是分析

- 倒排索引：索引表中的每一项都包括一个属性值和具有该属性值的各记录的地址。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引(inverted index)。
- 分析：文档在建立倒排索引之前，es让每个被分析字段所做的一系列操作
 - 字符过滤：使用字符过滤器转变字段，如大写转小写、&变成and等
 - 分词：将文本切分为单个或者多个词
 - 分词过滤：使用分词过滤器，转变每个分词
 - 分词索引：把这些分词和指向文档的关系放进索引

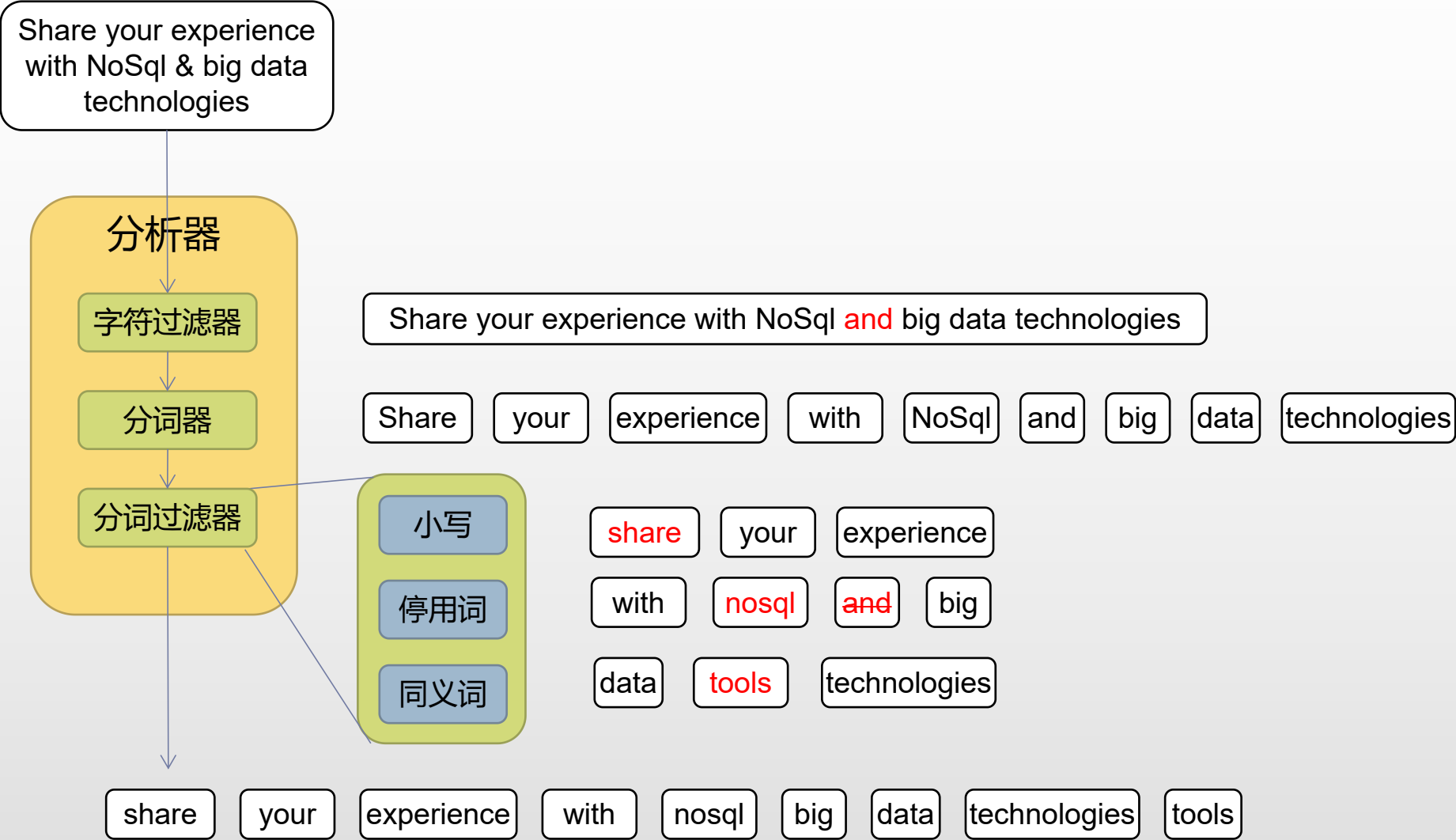


正向索引



倒排索引

分析图例



配置分析器

- 创建索引的时候配置分析器
- 使用template配置分析器
- 在elasticsearch的配置里设置全局默认分析器
- 全文检索类的搜索语句可以指定分析器，优先级如下
 - query参数里指定的
 - 被搜字段的search_analyzer指定的
 - 被搜字段的analyzer指定的
 - index配置里default_search指定的
 - index配置里default指定的
 - standard analyzer

```
PUT my_index
{
  "settings": {
    "analysis": { 分析器设置
      "analyzer": { 定制分析器
        "my_custom_analyzer": { 定制的分析器名称
          "type": "custom",
          "char_filter": [ 指定字符过滤器
            "emoticons" ❶
          ],
          "tokenizer": "punctuation", ❷ 指定分词器
          "filter": [ 指定分词过滤器
            "lowercase",
            "english_stop" ❸
          ]
        }
      },
      "tokenizer": { 自定义分词器
        "punctuation": { ❹
          "type": "pattern",
          "pattern": "[ .,!?]"
        }
      },
      "char_filter": { 自定义字符过滤器
        "emoticons": { ❺
          "type": "mapping",
          "mappings": [
            ":) => _happy_",
            ":( => _sad_"
          ]
        }
      },
      "filter": { 自定义分词过滤器
        "english_stop": { ❻
          "type": "stop",
          "stopwords": "_english_"
        }
      }
    }
  }
}
```


使用分析api

- `_analyze` api

- 对指定字符串使用指定分析器进行分析，直接展示分析结果
- 可以指定各种预定义分析器、自定义分析器，
- 甚至可以分别指定字符过滤器、分词器、分词过滤器

- https://www.elastic.co/guide/en/elasticsearch/reference/6.8/testing_analyzers.html

- `_termvectors` api

- 查看某个具体的文档的具体索引信息
- 这个文档有哪些分词，以及每个分词的词频、位置、开始和结束位置等

- <https://www.elastic.co/guide/en/elasticsearch/reference/6.8/docs-termvectors.html>

```
GET robin-test/_doc/3/_termvectors
{
  "fields" : ["a","my_text"],
  "offsets" : true,
  "payloads" : true,
  "positions" : true,
  "term_statistics" : true,
  "field_statistics" : true
}
```

```
5  "_version": 1,
6  "found": true,
7  "took": 0,
8  "term_vectors": {
9    "a": {
10     "field_statistics": {
11       "sum_doc_freq": 9,
12       "doc_count": 1,
13       "sum_ttf": 9
14     },
15     "terms": {
16       "bone": {
17         "doc_freq": 1,
18         "ttf": 1,
19         "term_freq": 1,
20         "tokens": [
21           {
22             "position": 10,
23             "start_offset": 51,
24             "end_offset": 55
25           }
26         ]
27       },
28     }
```

分析器

- Standard Analyzer
 - 标准分析器，如果不指定就是默认使用
 - 它删除大多数标点符号、字母大写转小写，并删除停用词
- Simple Analyzer
 - 遇到非字母就分词、所有字母转小写
- Whitespace Analyzer
 - 只是按空格分词，别的啥也没干
- Stop Analyzer
 - 在simple基础上，过滤掉停用词
- Keyword Analyzer
 - 真的什么也没干，就把字段当作关键词
 - 最好别用，直接用keyword类型不要分析字段就好了
- Pattern Analyzer
 - 可以配置正则表达式做为分词条件
 - 此外还会转小写和删除停用词
- Language Analyzers
 - 多语言分词器，用于特定语言的字符串
 - 看了一下，支持34种语言
 - 中文？不存在的
- Fingerprint Analyzer
 - 生成指纹
 - 一般用来检测字段重复
 - 比如论文查重
- Custom Analyzer
 - 自定义分析器
 - 可以随便组合内置的或者你自己定义的字符过滤器、分词器、分词过滤器

分词器 - Word Oriented Tokenizers

- Standard Tokenizer
 - 标准分词器，标准分析器里用的，基于语法的分词器
 - 大致是按空格、标点切分，适用于大多数欧洲语言
- Letter Tokenizer
 - 字母分词器，只要遇到不是字母的就分词，数字、符号、标点、空格等等
- Lowercase Tokenizer
 - 相当于字母分词器+小写过滤器
- Whitespace Tokenizer
 - 空格分词器，遇到空格、制表符、换行等空白符号则分词，注意不会去掉标点
- UAX URL Email Tokenizer
 - 在标准分词器基础上，增加对url和邮箱的识别
 - 比如Email me at john.smith@global-international.com切为Email, me, at, john.smith@global-international.com
- Classic Tokenizer
 - 专门针对英语的分词器，可以识别缩略词、公司名、邮箱、网络地址
- Thai Tokenizer
 - 泰语分词器，专门针对泰语，如果不是泰语则变为标准分词器

分词器 - Partial Word Tokenizers

- N元语法和侧边N元语法分词器，是es中非常独特的分词器，可以把单词切分为多个片段，以便于部分匹配。
- N-Gram Tokenizer
 - N元语法分词器，先把单词按空格、标点等切成单词，再把单词切成n个字符的片段。
 - 可配置min_gram最小元，max_gram最大元，token_chars分词范围（可选字符、数字、空格、标点、符号）
 - 比如：{"min_gram": 1,"max_gram": 2,"token_chars": ["letter","digit"]}
 - 2 Quick Foxes. =>
 - 2 Q Qu u ui i ic c ck k F Fo o ox x xe e es s
 - 可用于模糊查询和未知语言分析
- Edge N-Gram Tokenizer
 - 是N元分析的变体，从一侧开始切词。
 - {"min_gram": 2,"max_gram": 10,"token_chars": ["letter","digit"]}
 - 2 Quick Foxes. =>
 - Qu, Qui, Quic, Quick, Fo, Fox, Foxe, Foxes
 - 可以用于按每个单词，做类似前缀匹配的搜索

分词器 - Structured Text Tokenizers

- 用于处理一些结构化的字段，比如证件号、邮箱、邮编、路径等等
- Keyword Tokenizer
 - 关键词分词器，什么都没干，把整个语句当作一个分词。
- Pattern Tokenizer
 - 模式分词器，可以指定类似正则表达式的方式进行分词，非常灵活。
- Simple Pattern Tokenizer
 - 简化的模式分词器，表达式比较有限，运算速度稍微快一点点。
- Simple Pattern Split Tokenizer
 - 类似简化模式分词器，不同之处在于匹配中的短语是作为分隔符，而不是分词。
- Path Tokenizer
 - 路径分词器，专门切路径的，比如
 - `/foo/bar/baz` → `[/foo, /foo/bar, /foo/bar/baz]`.

分词过滤器

- Standard Token Filter
 - 啥也没干
- ASCII Folding Token Filter
 - 把非ASCII字符映射成等同的ASCII字符，前提是有等同字符存在，如ü转乘u
- Flatten Graph Token Filter
 - 将graph token流扁平化，配合 Synonym Graph Token Filter使用
- Length Token Filter
 - 长度过滤器，把设置范围长度外的分词过滤掉
- Lowercase Token Filter
 - 字母全部转小写
- Uppercase Token Filter
 - 字母全部转大写
- NGram Token Filter
 - n元语法过滤器，类似侧边n元语法分词器
- Edge NGram Token Filter
 - 侧边n元语法过滤器，类似侧边n元语法分词器
- Porter Stem Token Filter
 - 提取词干
- Shingle Token Filter
 - 滑动窗口分词过滤器，类似N元语法，只不过它处理整个分词
- Stop Token Filter
 - 过滤掉停用词，可以指定语言
- Word Delimiter Token Filter
 - 根据定界符拆词，比如"Wi-Fi" → "Wi", "Fi", 还可以"SD500" → "SD", "500"
- Word Delimiter Graph Token Filter
 - 搜索时用的，根据定界符拆词
- Stemmer Token Filter
 - 也是用来提取词干，可以配置各种语言
- Stemmer Override Token Filter
 - 通过自定义映射的方式来覆盖词干算法，然后保护这些术语不被stemmer修改
- Keyword Marker Token Filter
 - 保护单词不被stemmer修改
- Keyword Repeat Token Filter
 - 保留一份被stemmer修改的单词的原始词，比如 "cats" → "cats", "cat"

分词过滤器

- KStem Token Filter
 - 提取词干
- Snowball Token Filter
 - 提取词干
- **Synonym Token Filter**
 - 映射同义词
- Synonym Graph Token Filter
 - 搜索时用的，映射同义词
- Compound Word Token Filters
 - 复合词拆解
- **Reverse Token Filter**
 - 把单词颠倒，比如 "cats" → "stac"，用于想把*通配符放前面的搜索
- Elision Token Filter
 - 处理省略音标记，比如 "l'avion" → "avion"
- **Truncate Token Filter**
 - 用于根据配置的长度截断分词
- **Unique Token Filter**
 - 删除重复的分词
- Pattern Capture Token Filter
 - 用正则匹配的方式分词
- Pattern Replace Token Filter
 - 将分词正则替换
- **Trim Token Filter**
 - 前后去空格
- Limit Token Count Token Filter
 - 限制分词的数量
- Hunspell Token Filter
 - 提取词干
- Common Grams Token Filter
 - 对停用词用二元语法生成分词，"the quick brown is a fox" → "the", "the_quick", "quick", "brown", "brown_is", "is_a", "a_fox", "fox"
- Normalization Token Filter
 - 规范化一些语言的特殊字符，Arabic, German等等

分词过滤器

- CJK Width Token Filter
 - 将全角 ASCII 字符 转换为半角 ASCII 字符, CJK 是 Chinese, Japanese, Korean
- CJK Bigram Token Filter
 - 对中日韩文用二元语法生成分词, 因为不知道怎么切
- Delimited Payload Token Filter
 - 可以识别设定的分词的权重, 比如 "the|1 quick|2 fox|3" → "the", "quick", "fox",
- Keep Words Token Filter
 - 只保留指定的分词, 需要具体列出
- Keep Types Token Filter
 - 只保留指定类型的分词, 比如只要数字
- Classic Token Filter
 - 专门用来处理 classic tokenizer 产生的分词
- Apostrophe Token Filter
 - 去掉撇号和其后的字符, 比如 "Bob's" → "Bob"
- Decimal Digit Token Filter
 - 将 unicode 数字转化为 0-9
- Fingerprint Token Filter
 - 生成指纹, 把所有分词排序, 去重, 合成单个分词, "the", "quick", "quick", "brown", "fox", "was", "very", "brown" → "brown fox quick the very was"
 - 我们可以用来合并一些相似的文本, 比如验价、生单的报错提示
- Minhash Token Filter
 - 生成分词的最小哈希, 通常用来判断重复

字符过滤器

- HTML Strip Character Filter
 - 去掉html标识符，并对html编码的符号解码
 - "<p>I'm so happy!</p>" → "\nI'm so happy!\n"
- Mapping Character Filter
 - 读取预先配置的字符映射表，对文本做字符映射
- Pattern Replace Character Filter
 - 对文本做正则替换

五、相关性计算

打分是如何运作的

- Elasticsearch 6开始，默认使用BM25算法进行打分
- TF-IDF
 - 词频-逆文档频率（Term Frequency-Inverse Document Frequency）是一种用于信息检索与文本挖掘的常用加权算法。它是一种统计方法，用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度。
 - 字词的重要性随着它在文件中出现的次数成正比增加，但同时会随着它在语料库中出现的频率成反比下降。
- Okapi BM25
 - 当两篇描述“人工智能”的文档A和B，其中A出现“人工智能”100次，B出现“人工智能”200次。两篇文章的单词数量都是10000，那么按照TF-IDF算法，A的tf得分是：0.01，B的tf得分是0.02。得分上B比A多了一倍，但是两篇文章都是再说人工智能，tf分数不应该相差这么多。可见单纯统计的tf算法在文本内容多的时候是不可靠的
 - 多篇文档内容的长度长短不同，对tf算法的结果也影响很大，所以需要将文本的长度也考虑到算法当中去
- https://segmentfault.com/a/1190000019630099?utm_source=tag-newest

$$score(t, q, d) = \sum_t^n (idf(t) * boost(t) * tfNorm(t, d))$$

$$idf_t = \ln \left(1 + \frac{docCount - docFreq + 0.5}{docFreq + 0.5} \right)$$

$$tfNorm(t, d) = \frac{f(t, d) \cdot (k_1 + 1)}{f(t, d) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{avgdl} \right)}$$

使用explain解释一次查询请求

- 在查询中加explain参数
- 访问一个文档的 explain接口

GET get-together-group/_search

```
{
  "query": {
    "match_phrase": {
      "description": {
        "query": "learn about",
        "slop": 1
      }
    }
  },
  "explain": true
}
```

```
GET get-together-group/_doc/1/_explain
```

```
{
  "query": {
    "match_phrase": {
      "description": {
        "query": "learn about",
        "slop": 1
      }
    }
  }
}
```

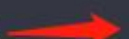

```

"max_score": 0.53131896,
"hits": [
{
  "_shard": "[get-together-group][2]",
  "_node": "mILa70cjROKs5GkCT7QrLg",
  "_index": "get-together-group",
  "_type": "_doc",
  "_id": "2",
  "_score": 0.53131896,
  "_source": {
    "name": "Elasticsearch Denver",
    "organizer": "Lee",
    "description": "Get together to learn more about using Elasticsearch, the applications and neat things you can do with ES!",
    "created_on": "2013-03-15",
    "tags": [
      "denver",
      "elasticsearch",
      "big data",
      "lucene",
      "solr"
    ],
    "members": [
      "Lee",
      "Mike"
    ],
    "location_group": "Denver, Colorado, USA"
  },
  "explanation": {
    "value": 0.53131896,
    "description": ""weight(description:"learn about"~1 in 0) [PerFieldSimilarity], result of:"",
    "details": [
      {
        "value": 0.53131896,
        "description": "score(doc=0,freq=0.5 = phraseFreq=0.5\n), product of:",
        "details": [
          {
            "value": 0.87546873,
            "description": "idf(), sum of:",
            "details": [
              {
                "value": 0.18232156,
                "description": "idf, computed as log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",
                "details": [
                  {
                    "value": 2,
                    "description": "docFreq",
                    "details": []
                  },
                  {
                    "value": 2

```

使用再打分机制来减小评分的性能损耗

- 有时打分会十分消耗资源
 - 使用脚本计算得分
 - 进行phrase查询，使用很大的slop值
 - 使用了通配查询
- 再打分(rescore)机制
 - 初始查询运行后，针对返回的结果进行第二轮的打分计算
 - 对于非常消耗性能的打分，应该放在rescore里面处理
 - 可以连续运行多个rescore，将会逐个运算

```
POST /_search
{
  "query" : {  初始查询
    "match" : {
      "message" : {
        "operator" : "or",  把带三个关键词中任意一个的文档查出来
        "query" : "the quick brown"
      }
    }
  },
  "rescore" : [ {
    "window_size" : 100,  取前100条再打分
    "query" : {
      "rescore_query" : {
        "match_phrase" : {  运行较耗资源的短语查询
          "message" : {
            "query" : "the quick brown",
            "slop" : 2
          }
        }
      }
    },
    "query_weight" : 0.7,
    "rescore_query_weight" : 1.2
  } ],
  "window_size" : 10,  取上一个结果中的前10名
  "query" : {
    "score_mode": "multiply",
    "rescore_query" : {
      "function_score" : {
        "script_score" : {  运行较耗资源的脚本打分
          "script" : {
            "source": "Math.log10(doc.likes.value + 2)"
          }
        }
      }
    }
  }
}
```

定制子查询的得分

- Boosting

- 索引期间可以通过设置字段mapping的boosting参数，对某个字段增加权重
 - 如果要修改这个权重，必须reindex
 - 用低精度的浮点存储，计算时可能会丢失精度
 - 如果被boost的字段中，匹配上了多个词条，意味着多次boost，每次都会加权

- 查询期间

- 几乎所有的查询类型都可以通过参数配置boosting
- 每个子查询都可以设置权重

- Function_score

- 是一种复合查询类型，可以对每个子查询的得分进行计算
- 基本运算：multiply, sum, avg, first, max, min
- 复杂运算：脚本、随机、科学运算、衰减
- <https://www.elastic.co/guide/en/elasticsearch/reference/6.8/query-dsl-function-score-query.html>

```
{
  "query": {
    "function_score": {
      "query": {
        "match_all": {}
      },
      "functions": [
        {
          "weight": 1.5,
          "filter": {
            "term": {
              "description": "hadoop"
            }
          }
        },
        {
          "field_value_factor": {
            "field": "reviews",
            "factor": 10.5,
            "modifier": "log1p"
          }
        },
        {
          "script_score": {
            "script": {
              "params": {
                "myweight": 3
              },
              "source": "Math.log(doc['attendees.keyword'].values.size() * params.myweight )"
            }
          }
        },
        {
          "gauss": {
            "location_event.geolocation": {
              "origin": "40.018528,-105.275806",
              "offset": "100m",
              "scale": "2km",
              "decay": 0.5
            }
          }
        }
      ]
    },
    "score_mode": "sum",
    "boost_mode": "replace"
  }
}
```

原始的查询匹配了所有文档

使用权重函数，把描述包含hadoop的文档，加权1.5倍

评论多的文档排名更高

参与人数多的排名更高

离指定位置越远，衰减越大

把每个函数的得分相加

替换原有的match_all查询的得分

字段数据和缓存

- 其实和算分没有太大关系，只是提到了脚本顺便说一下
- 这里指的是未被分析的字段的数据，用到的场景有：
 - 对字段排序
 - 对字段聚集
 - 脚本中doc['fieldname']访问字段值
 - Function_score查询中使用field_value_factor函数或者decay函数
 - 在搜索请求中指定fielddata_fields获取字段内容
- 字段数据会被大规模加载，因此elasticsearch会将其缓存到内存
 - 查询时加载和预热器加载
- 加载太多了会占用过多内存
 - 限制内存占用量：可配置上限、缓存过期时间，通过LRU原则淘汰数据
 - 断路器：加载之前预估，如果超过设定阈值，会直接抛出异常，避免无效加载，而且可动态配置
 - 使用文档值：索引建立的时候创建，访问的时候直接读取，索引会变慢，占磁盘更大，但加载更快，且不会OOM

六、聚集

什么是聚集

- 大概可以理解为分类统计，比如对一组数据的某个词条进行计数、或者计算某个数值型字段的平均值
- 在kibana上随处可见
 - 各种visualize都是基于此
- 分为度量聚集和桶聚集
- 对比搜索最大的不同
 - 不能使用倒排索引，需要用到前一节所说的字段数据
 - 聚集时会把倒排索引反转回字段数据，塞进内存，因此如果聚集操作频繁，就需要大量内存
- 后过滤器
 - 正常情况下过滤查询是先执行的，聚集在此基础上运行
 - 有时候需要先对所有数据进行聚集，再过滤查询出一些数据展示
 - 后过滤器是在聚集之后运行，和聚集操作相对独立，需要注意性能

度量聚集

- Avg Aggregation: 求平均值
- Weighted Avg Aggregation: 带权重的平均值
- Max Aggregation: 最大值
- Min Aggregation: 最小值
- Sum Aggregation: 求和
- Value Count Aggregation: 计数, 最常用
- Stats Aggregation: 一次性返回avg,max,min,sum,count
- Extended Stats Aggregation:
 - avg,max,min,sum,count
 - sum_of_squares, variance, std_deviation ,std_deviation_bounds
- Geo Bounds Aggregation: 求坐标边界, 返回矩形的左上和右下坐标
- Geo Centroid Aggregation: 求坐标中心, 返回一个点
- Scripted Metric Aggregation: 自定义聚集, 有点像map-reduce

```
GET get-together-event/_search
{
  "query": {
    "match_all": {}
  },
  "aggs": {
    "myAgg": {
      "stats": {
        "field": "date"
      }
    }
  }
}

{
  "took": 0,
  "timed_out": false,
  "_shards": { ...
  },
  "hits": { ...
  },
  "aggregations": {
    "myAgg": {
      "count": 15,
      "min": 1361212200000,
      "max": 1378751400000,
      "avg": 1371268440000,
      "sum": 20569026600000,
      "min_as_string": "2013-02-18T18:30:00.000Z",
      "max_as_string": "2013-09-09T18:30:00.000Z",
      "avg_as_string": "2013-06-15T03:54:00.000Z",
      "sum_as_string": "2621-10-22T10:30:00.000Z"
    }
  }
}
```

近似计算

- 普通的聚集操作都要全部遍历查询范围内的所有文档，如果数据量巨大时，需要很昂贵的代价，尤其是内存。很多时候并不需要精确的统计，可以牺牲部分精确性，来节省消耗的资源。
- Cardinality Aggregation：
 - 基数聚集，可以近似计算指定字段的distinct值
 - 用HyperLogLog++算法，对求基数的字段取散列（类似抽样）
 - 可以配置抽样大小
- Percentiles Aggregation：
 - 百分位聚集，得到n值，使百分之x的数据低于n值
 - 比如计算大多数用户购买商品的价格区间
- Percentile Ranks Aggregation
 - 和上面那个相反，给定n值，求x值
- Top Hits Aggregation：
 - 排序后分页展示结果，一般用于桶聚集的嵌套

桶聚集

- 把数据按照某个标签分组，**kibana**里非常常见
- Terms Aggregation：每个不同的词条一个桶
 - Significant Terms Aggregation：聚集显著词条
 - Significant Text Aggregation：聚集显著分词
- Range Aggregation：可以按指定范围分桶，通常用于数值类型字段
- Date Range Aggregation：指定时间范围分桶
- Histogram Aggregation：类似range，但是间隔固定长度，用于绘制直方图
- Date Histogram Aggregation：日期直方图分桶
- **桶下可以嵌套桶或者数值**

七、关系

不推荐使用

- Es有提供文档关系的操作，但有很多限制，功能也不是很强大
- 后期版本把_type去掉之后，文档的关系操作就更加弱化了
 - 本来同一个index下不同的type的文档，可以建立关系，并通过某种路由机制把有关系的文档存在同一个分片

八、提升性能

提升写入性能

- 用bulk接口批量写入
 - 可以节省重复创建连接的网络开销
 - 要通过测试才能知道最佳的一次批处理量，并不是越大越好，太大了会占用内存
 - 并且bulk有个处理队列，过慢的index会导致队列满而丢弃后面的请求
- 配置慢一点的刷新频率
 - es是准实时系统，新写入的分段需要被刷新才被完全创建，才可用于查询
 - 慢的刷新频率可以降低分段合并的频率，分段合并十分耗资源
 - 默认刷新频率是1s，对index修改index.refresh_interval即可立即生效
- 初始化性质的大量写入
 - 比如reindex或是导入基础数据这种一次性批量索引操作
 - 可以配置成不刷新，并且把副本数也配置成0，完了之后再设置成正常值
 - 每一次写入都要等所有副本都报告写入完成才算完成，副本数量越多写入越慢
- 关闭操作系统的swapping
 - 操作系统会自动把不常用的内存交换到磁盘（虚拟内存）
 - es是运行于jvm的，这个操作可能会导致gc
- 使用内部id
 - 默认是指明文档id的，这样的话es需要先判断一下这个id的文档是否已经存在，以做一些合并或者更新操作
 - 如果用自生成的id，则可以跳过这个步骤节省开支
- 合理设置分片和副本数量
 - 分片数量影响到分段数量，分片少的话允许的分段数也会少，从而会增加分段合并的频率，消耗性能
 - 如果写入规模巨大，要控制index的规模（按月、按周、按天适当分，或自动滚动），同时根据集群节点数量设置合适的分片数，使得每个分片的数据量有限
 - 副本数量越多，写入越慢
- 合理设置字段mapping
 - 不需要分析的字段就不要分析

提升查询性能

- 用过滤器上下文
 - 不计算得分可以减少资源消耗
 - 过滤器还可以缓存
- 避免脚本
 - 脚本非常耗性能，因为每次计算且无法缓存
 - 如果非用不可，用painless或者expressions
- 提前索引字段
 - 比如某个字段经常被range查询或聚集
 - 那在索引字段的时候，就把range范围确定好
 - 比如15属于10-100，就存一个15一个10-100
- 合理mapping
 - 使用适当的分析器，如果对查询速度要求很高，就要在索引的时候牺牲性能
 - 数字是存在另外的地方，所以有时候数字可以存成keyword而不是numeric会更快
- 有意识地使用更轻量的查询语句
 - 比如term查询比query查询更省资源，query会被分析，衍生出很多子查询
 - 通配符查询很费性能，尤其是通配符放在很前面
- 不要使用任何的关联关系
 - 不管是嵌套还是父子，都会使查询量倍增
 - 通过冗余数据，以空间换时间，存储的成本很低
- 增加副本数量
 - 可以均衡查询负载
- 分配感知
 - 如果es按时间分索引，你又恰好知道它在哪个时间段，精确的查询到这个索引而不是查一大片，显然会更快
 - 索引的时候有一个_route参数，可以控制某个文档索引到哪个分片，如果你的一个查询的所有结果都从一个分片获取，就能减少数据合并的开销，如果分片在不同机器，还能节省网络开销
 - 节点的配置有一个allocation awareness，可以根据rack、group、zone来配置节点，使得分片均匀分布，从而降低单点热度，同一个分片的副本不在一起，还可以容灾
- 按时间查询的时候对时间取整
 - 可以更容易命中缓存
- 如果index不再写入可以合并分段
 - 分段越少，查询越快，因为每次查询都要拆到所有分段去处理，再合并结果
 - 有一个_forcemerge接口，可以把分段弄成1
 - 同理，甚至可以合并分片（reindex或shrink）
- 给文件系统预留足够内存
 - 机器内存最多分一半给es，剩下留给文件系统
 - 因为es非常依赖操作系统的文件缓存，尤其是查询操作
- 用ssd磁盘而且别用远程
 - es需要频繁读取磁盘

节省磁盘空间

- 关闭不需要的mapping特性
 - 不被用来查询的字段，不索引
 - 不做全文检索，不分词(keyword)
 - 不关注文档相关性，关闭norms
 - <https://www.elastic.co/guide/en/elasticsearch/reference/6.8/norms.html>
 - 不需要短语检索，关闭位置索引
- 不要使用自动mapping
 - 默认会对string字段做两次索引(text和keyword)
- 留意分片大小
 - 分片越大，存储效率越高
 - 滚动存储，使其大小可控
 - 使用收缩api收缩分片
- 关闭不用的字段
 - `_all`
 - `_source`
- 配置压缩存储
- 分段合并
 - Force merge
- 数字类型的字段用最小类型
 - `byte < short < integer < long`

理解越深刻， 优化越透彻

Elasticsearch证书

- 据说含金量很高
- 考一次400刀
- 用你自己的电脑，远程考试
- 需要打开摄像头，全程不能用手机，有个人专门盯着你
- 大部分是如何解决问题这样的动手题
- 考试大纲恰好跟本课内容高度重合
- 报名链接：<https://training.elastic.co/exam/elastic-certified-engineer>

教学视频

- 作者B站主页
 - <https://space.bilibili.com/411333099>
- 教学视频（B站）
 - <https://www.bilibili.com/video/BV1kf4y1R7qR>
 - <https://www.bilibili.com/video/BV1h5411a7tC>
 - <https://www.bilibili.com/video/BV1EK411T7V1>
 - <https://www.bilibili.com/video/BV1jV411S7Pm>

谢谢