elastic

走近 ElasticSearch

基础服务中台 —基础架构部

▶ 主讲人：杨松柏

2021.05.31

目 录

PART 01

搜索的本质

01 搜索的本质

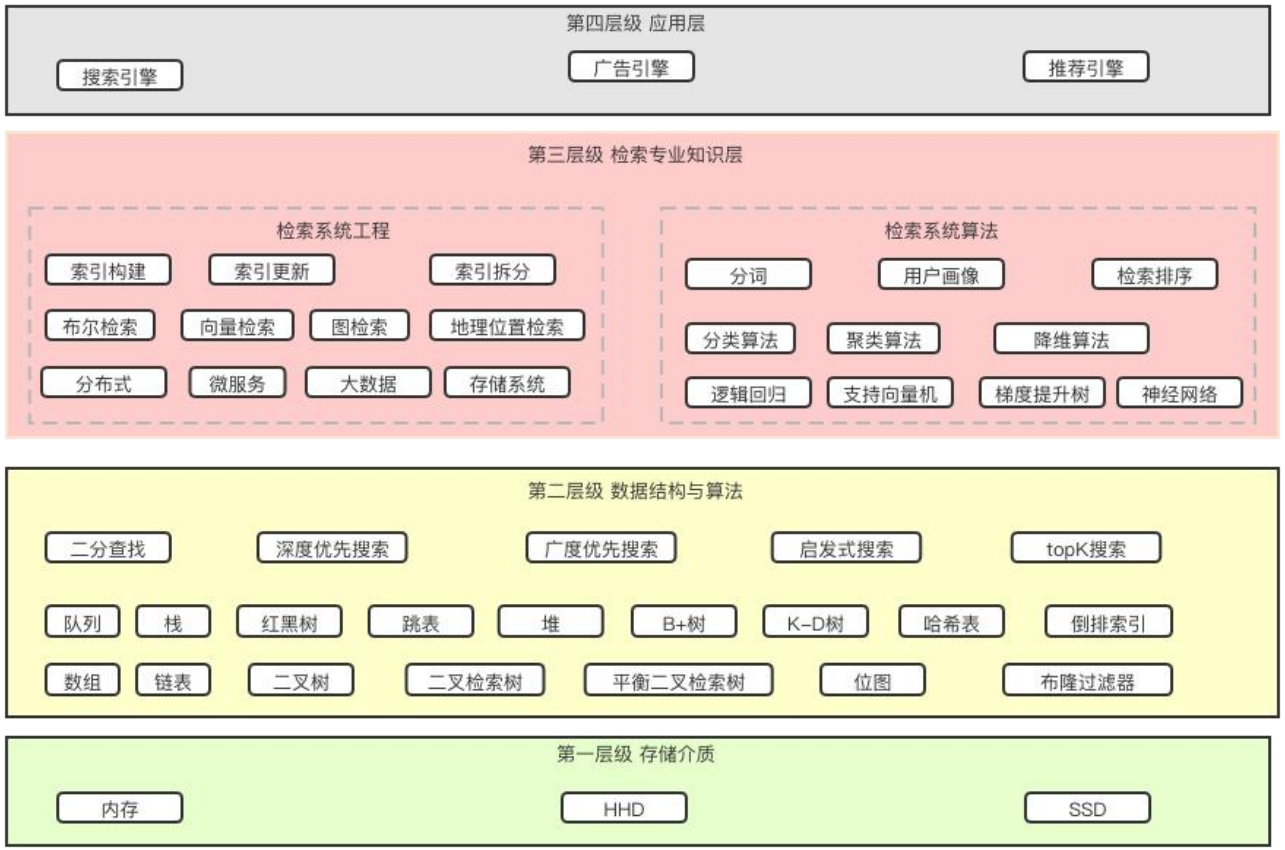
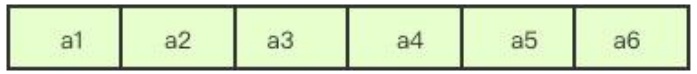


图 1-1 检索技术相关知识(取材自网络)

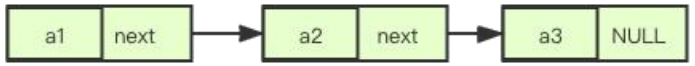
什么是检索？

检索其实就是将我们所需要的信息，从存储数据的地方高效取出的一种技术。所以，检索效率和数据存储的方式是紧密联系的。具体来说，就是不同的存储方式，会导致不同的检索效率。

01 搜索的本质

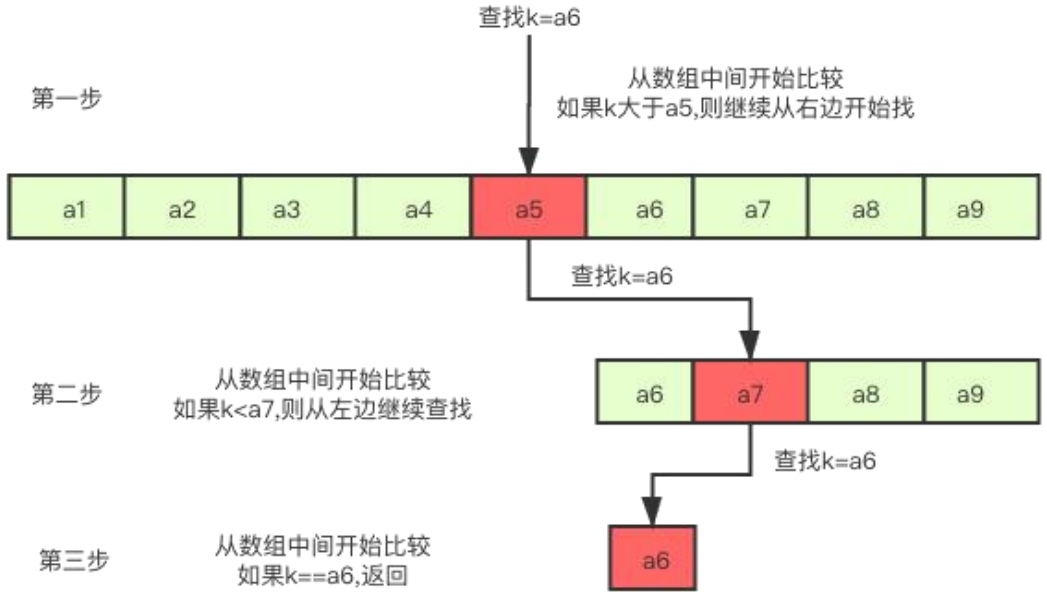


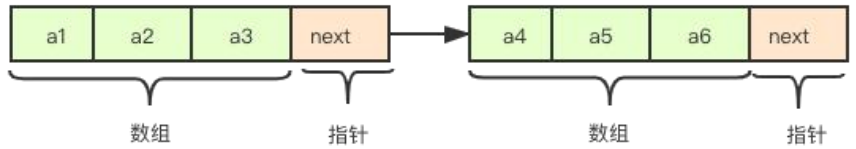
数组



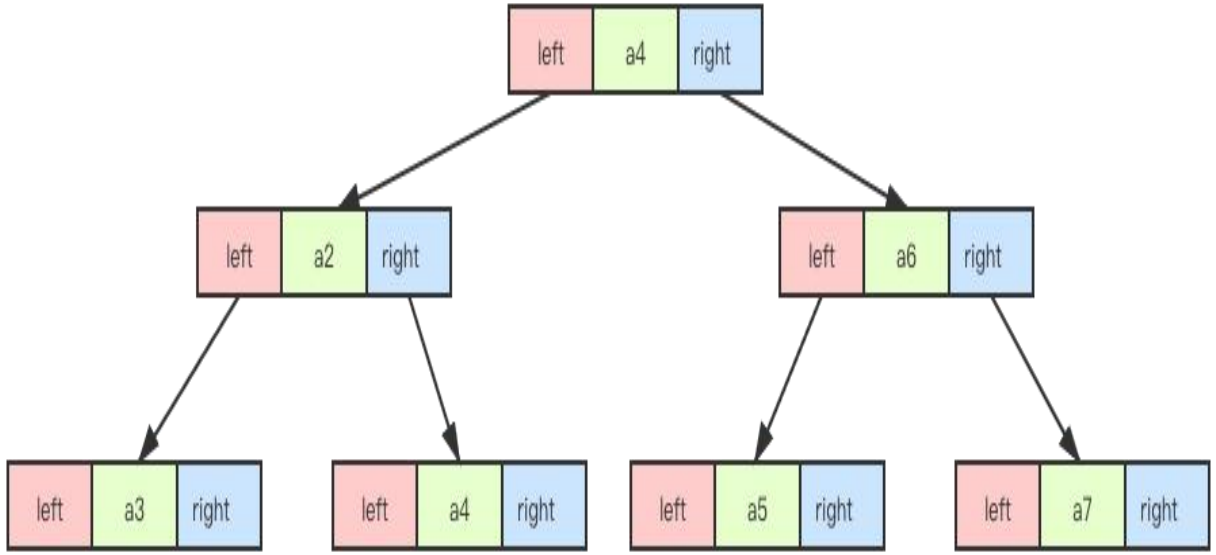
链表

数组和链表分别代表了连续空间和不连续空间的最基础的存储方式，
它们是线性表（Linear List）的典型代表。
其他所有的数据结构，比如栈、队列、二叉树、B+ 树等，
都不外乎是这两者的结合和变化

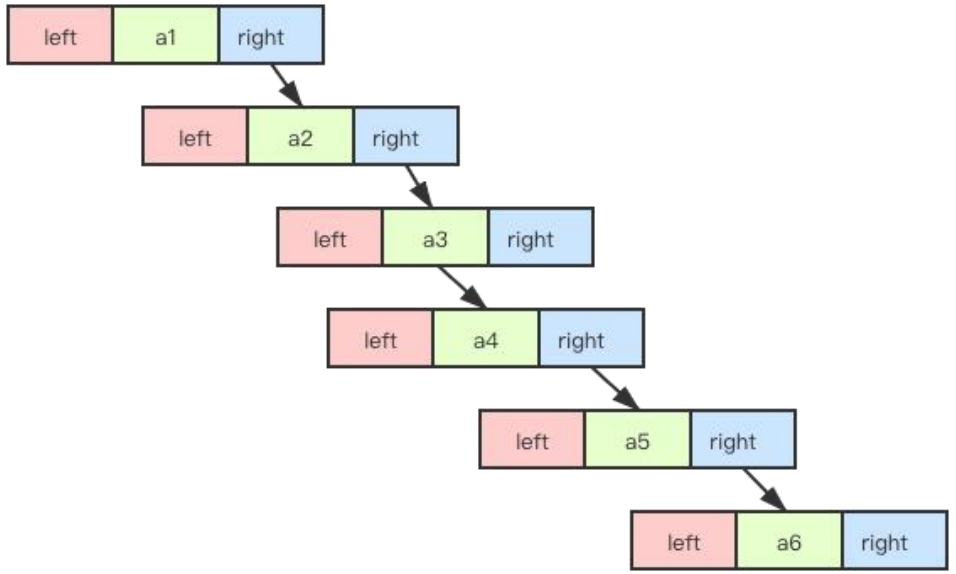




01 搜索的本质



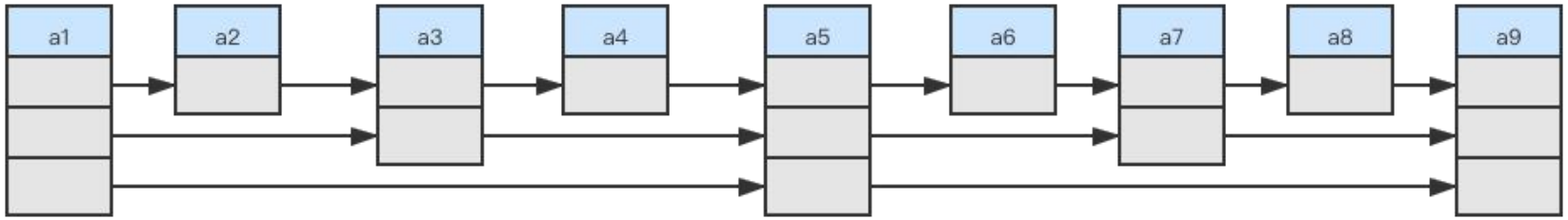
二叉搜索树



退化成链表

各种手段控制二叉搜索数据的 平衡性，如：AVL 树（平衡二叉树）和红黑树

跳表实现二分查找



01 搜索的本质

为了加快数据检索，我们可以为数据加建不同数据结构的索引

B树索引、哈希索引、位图索引、倒排索引等

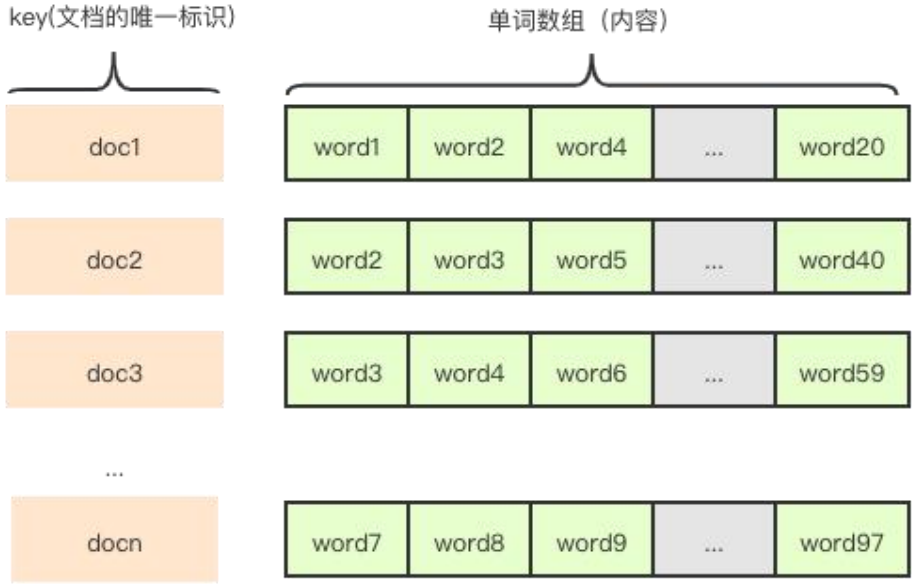
01 搜索的本质

试想这样一个场景：假设你是一位资深王者荣耀的玩家，那么你一定熟悉每位英雄的技能。
这个时候：
（1）如果我给你一个英雄的名字，你可以说出这个英雄是哪几种属性的吗（坦克/辅助/法师等等）？
（2）但是如果我问你，有哪些英雄同时包含“坦克”和“辅助”属性？

在回答第一个问题，你应该能立刻回答的出来的；
但是第二个问题你就不见得能立刻回答出来了。你需要在头脑中一个一个英雄的去回忆，并判断每个英雄的属性是否同时包含了“坦克”字和“辅助”方法。
很显然，第二个问题的难度比第一个问题大得多。

在技术上以上两个问题对应检索的正排索引和倒排索引

01 搜索的本质

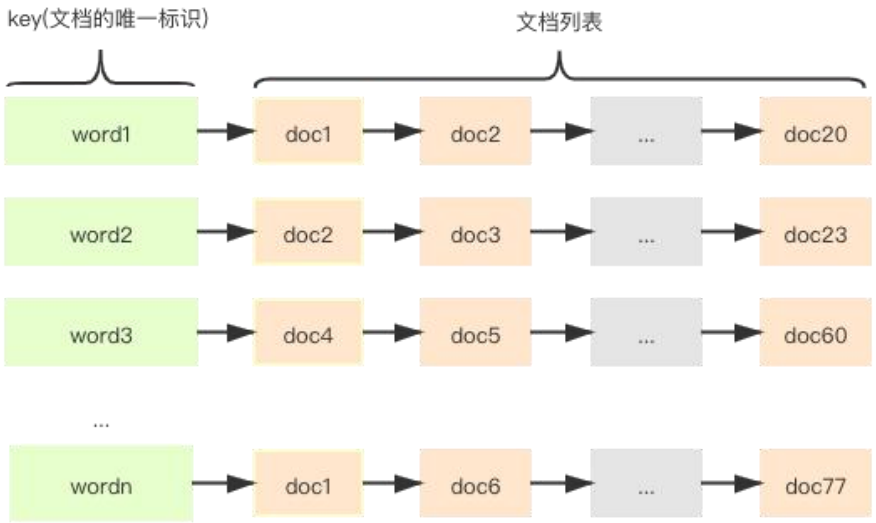


我们可以给每个英雄一个唯一的编号作为 ID，然后使用哈希表将英雄的 ID 作为键（Key），把英雄的属性作为键对应的值（Value）。
这样，我们就能在 O(1) 的时间代价内，完成对指定 key 的检索。
这样一个以对象的唯一 ID 为 key 的哈希索引结构，叫作正排索引（Forward Index）

一般来说，我们会遍历哈希表，遍历的时间代价是 O(n)。
在遍历过程中，对于遇到的每一个元素也就是每个英雄，我们需要遍历这英雄中的每一个属性，才能判断是否包含“坦克”字和“辅助”两个词。假设每个英雄的包含属性的长度是 k，那遍历一个英雄的时间代价就是 O(k)。从这个分析中我们可以发现，
这个检索过程全部都是遍历，因此时间代价非常高。对此，有什么优化方法吗？

我们先来分析一下这两个场景。我们会发现，“根据题目查找内容”和“根据关键字查找题目“,这两个问题其实是完全相反的。既然完全相反，那我们能否“反着”建立一个哈希表来帮助我们查找呢？也就是说，如果我们以关键字作为 key 建立哈希表，是不是问题就解决了呢？接下来，我们就试着操作一下。

01 搜索的本质

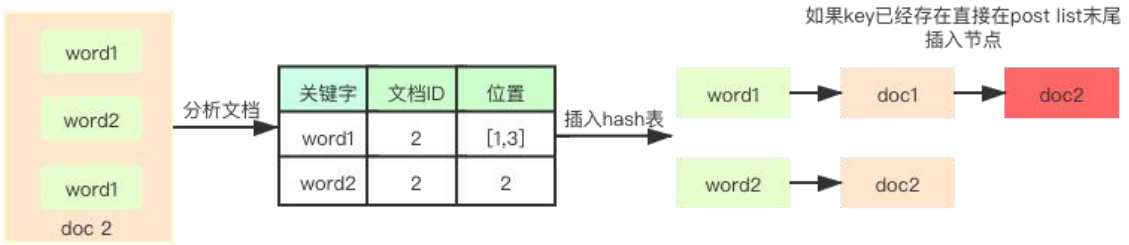


我们将每个关键字当作 key，将包含了这个关键字的诗的列表当作存储的内容。
这样我们就建立了一个哈希表，根据关键字来查询这个哈希表，在 O(1) 的时间内，
我们就能得到包含该关键字的文档列表。这种根据具体内容或属性反过来索引文档标题的结构，
我们就叫它**倒排索引**（Inverted Index）。在倒排索引中，key 的集合叫作**字典**（Dictionary），
一个 key 后面对应的记录集合叫作记录列表（Posting List）。

01 搜索的本质

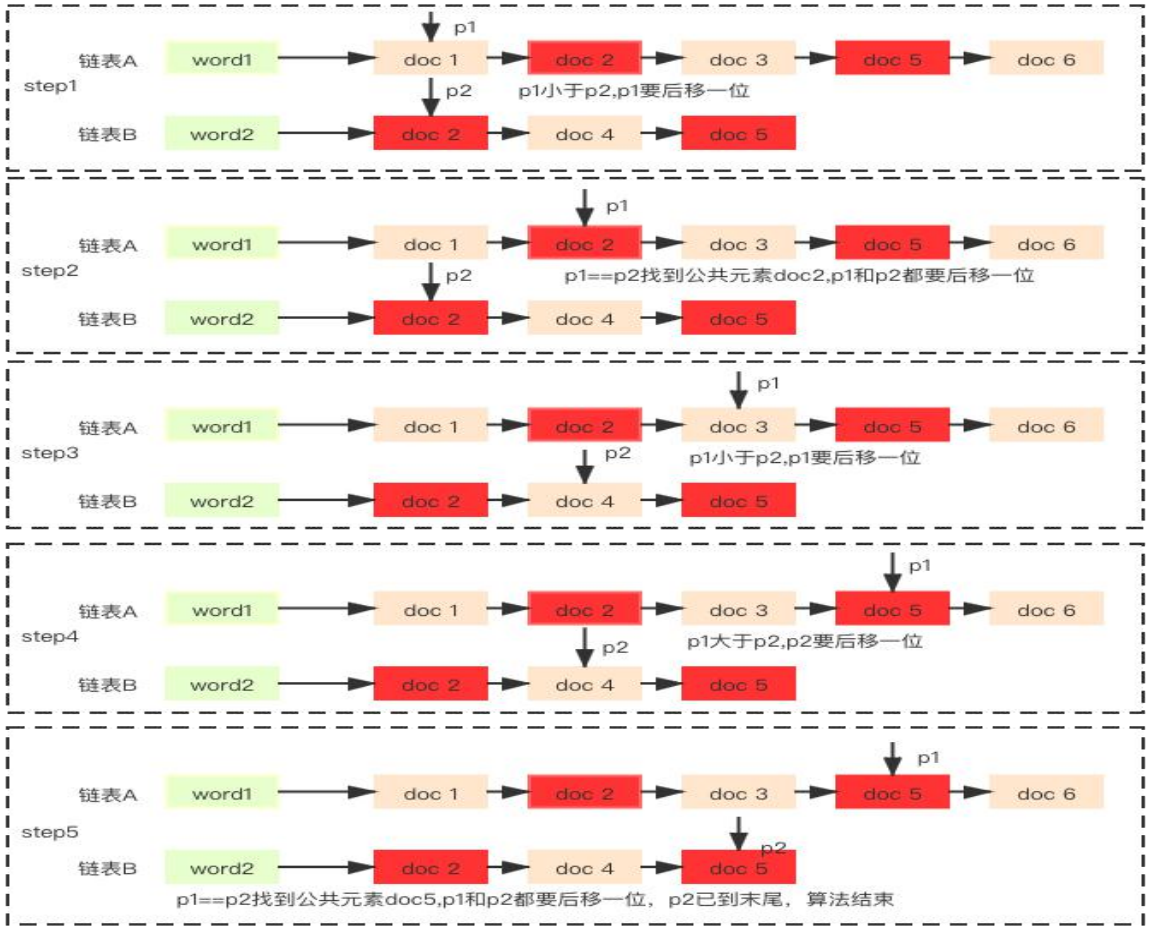
倒排索引的创建过程

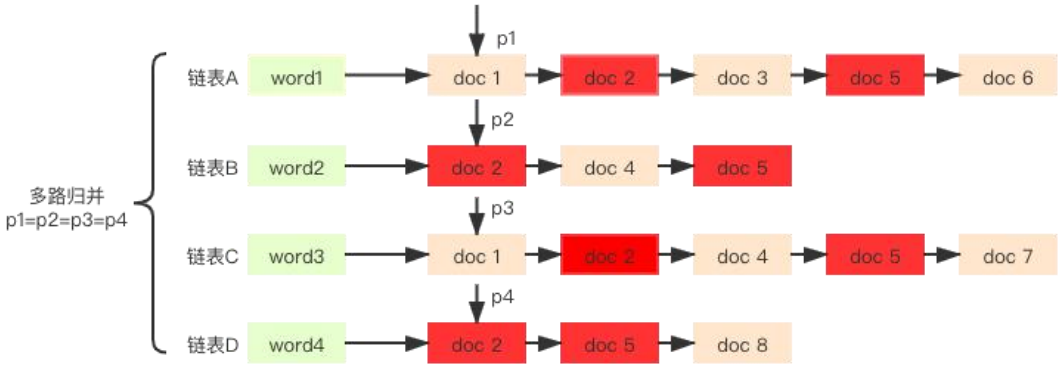
1. 给每个文档编号，作为其唯一的标识，并且排好序，然后开始遍历文档（为什么要先排序，然后再遍历文档呢？）。
2. 解析当前文档中的每个关键字，生成 **< 关键字，文档 ID，关键字位置 >** 这样的数据对。为什么要记录关键字位置这个信息呢？因为在许多检索场景中，都需要显示关键字前后的内容，比如，在组合查询时，我们要判断多个关键字之间是否足够近。所以我们需要记录位置信息，以方便提取相应关键字的位置。
3. 将关键字作为 key 插入哈希表。如果哈希表中已经有这个 key 了，我们就在对应的 posting list 后面追加节点，记录该文档 ID（关键字的位置信息如果需要，也可以一并记录在节点中）；如果哈希表中还没有这个 key，我们就直接插入该 key，并创建 posting list 和对应节点。
4. 重复第 2 步和第 3 步，处理完所有文档，完成倒排索引的创建。



01 搜索的本质

如何查询同时含有“坦克”字和“辅助”词两个 key 的文档？O(m + n)

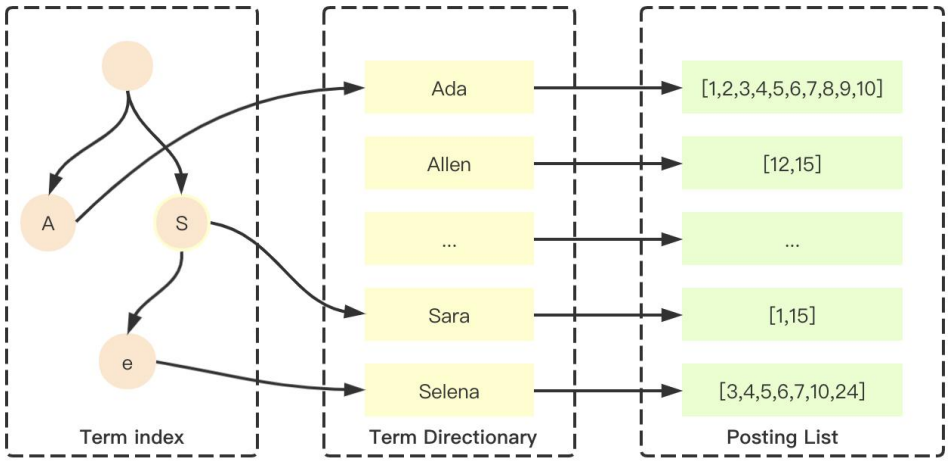




查找同时包含多个属性

工业界加速倒排索引的查找方法：
跳表、哈希表、位图、Roaring Bitmap等等

01 搜索的本质



将磁盘里的东西尽量搬进内存，减少磁盘随机读取次数(同时也利用磁盘顺序读特性)，
结合各种奇技淫巧的压缩算法，用及其苛刻的态度使用内存。

Roaring Bitmap
- postlings list
- numeric doc values

LZ4
- Stored fields, term vectors

FST
- terms index

PART 02

基本概念、倒排索引、分词、scale

02 基本概念、倒排索引、分词、scale

倒排索引在elasticsearch中的简单应用(实战)

02 基本概念、倒排索引、分词、scale

Analysis

文本分析是把全文本转换一系列单词(term/token)的过程，也叫分词。Analysis是通过**Analyzer来实现**的。
当一个文档被索引时，每个Field都可能会创建一个倒排索引（Mapping可以设置不索引该Field）。
倒排索引的过程就是将文档通过Analyzer分成一个一个的Term,每一个Term都指向包含这个Term的文档集合。

当查询query时，Elasticsearch会根据搜索类型决定是否对query进行analyze，然后和倒排索引中的term进行相关性查询，匹配相应的文档。

Analyzers

分析器（analyzer）都由三种构件块组成的：character filters,tokenizers,token filters

- **character filter 字符过滤器**

  在一段文本进行分词之前，先进行预处理，比如说最常见的就是，
  过滤html标签（好未来 --> 好未来），& --> and（I&you --> I and you）

- **tokenizers 分词器**

  英文分词可以根据空格将单词分开,中文分词比较复杂,可以采用机器学习算法来分词。

- **Token filters Token过滤器**

三者顺序：Character Filters--->Tokenizer--->Token Filter

三者个数：analyzer = CharFilters（0个或多个） + Tokenizer(恰好一个) + TokenFilters(0个或多个)

02 基本概念、倒排索引、分词、scale

elasticsearch内置分词器

- Standard Analyzer - 默认分词器，按词切分，小写处理
- Simple Analyzer - 按照非字母切分(符号被过滤), 小写处理
- Stop Analyzer - 小写处理，停用词过滤(the,a,is)
- Keyword Analyzer - 不分词，直接将输入当作输出
- Whitespace Analyzer - 按照空格切分，不转小写
- Language Analyzer - 提供了30多种常见语言的分词器
- Patter Analyzer - 正则表达式，默认\W+(非字符分割)
- Fingerprint Analyzer 文本会被转为小写格式,
经过规范化(normalize)处理之后移除扩展字符,然后再经过排序,
删除重复数据组合为单个token;如果配置了停用词则停用词也将会被移除
- Customer Analyzer 自定义分词器

开源分词插件

elasticsearch-analysis-ansj
elasticsearch-analysis-ik

elasticsearch-analysis-pinyin

jieba 分词器

02 基本概念、倒排索引、分词、相关性

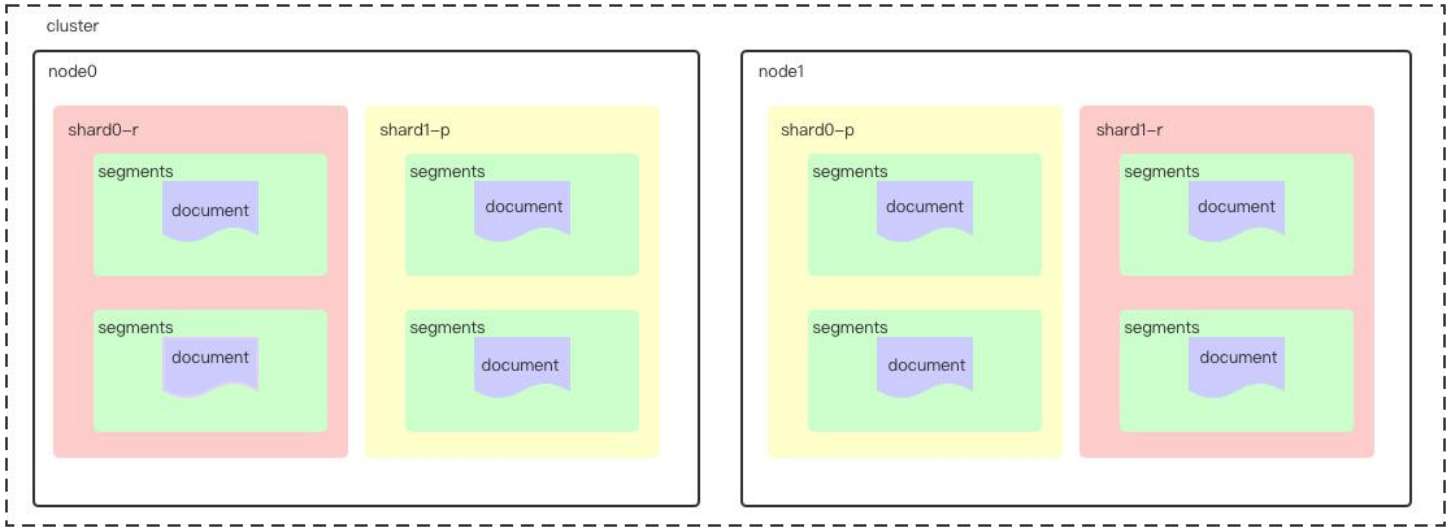
自定义分词器-实战

02 基本概念、倒排索引、分词、相关性

Elasticsearch模块架构

02 基本概念、倒排索引、分词、相关性

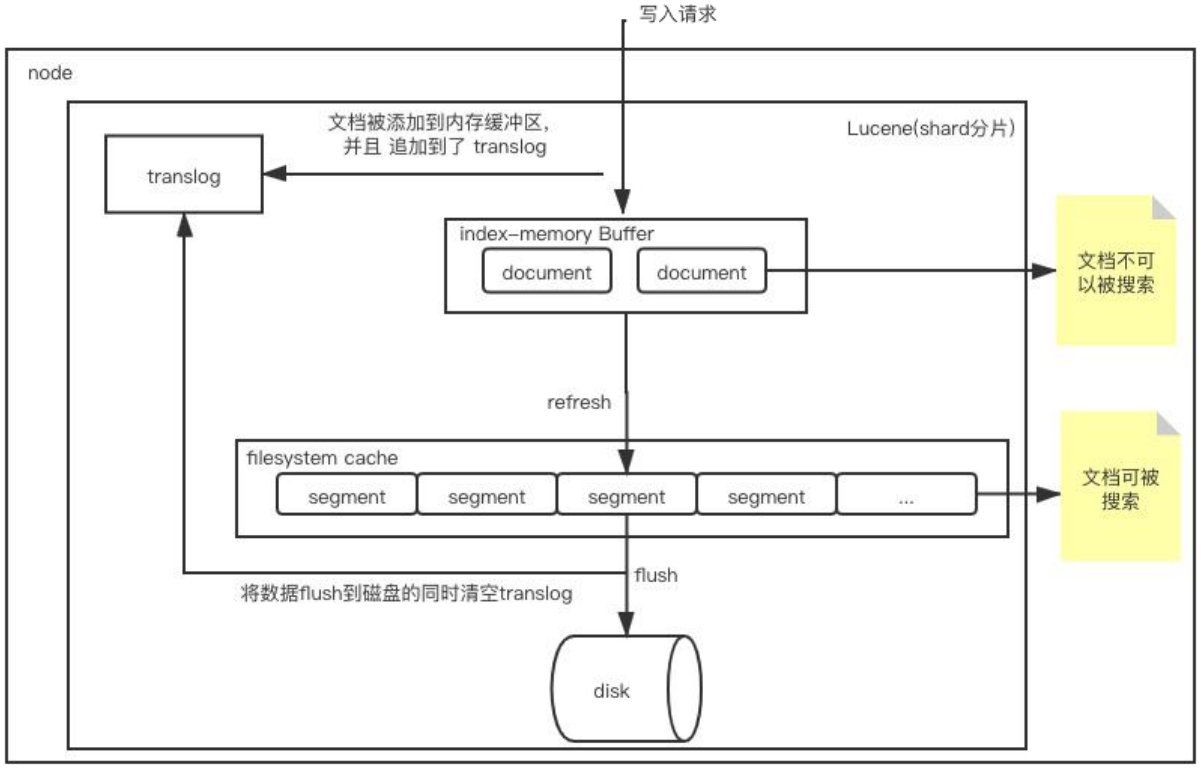
Elasticsearch基本概念



02 基本概念、倒排索引、分词、相关性

Elasticsearch、Lucene、segements

02 基本概念、倒排索引、分词、scale



实时性保证---refresh

"index.refresh_interval": "1s"

安全性保证---translog

"index.translog.durability": "async",
"index.translog.flush_threshold_size": "1024mb",
"index.translog.sync_interval": "120s"

大量segments的，会影响查询性能

进行semegent段的合并

合并成一个段（segment),就好吗？

02 基本概念、倒排索引、分词、相关性

segment内部包含的数据结构

- Inverted Index
- Stored Fields
- Document Values
- Cache

Segments是不可变的（immutable）

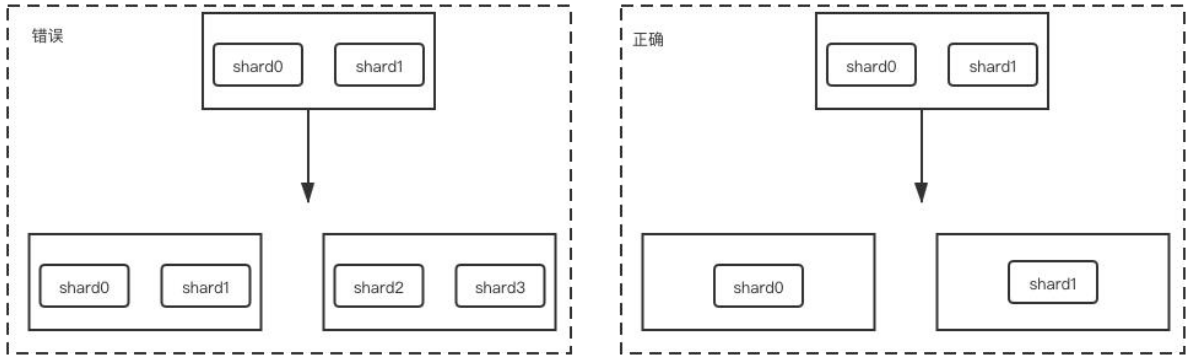
- 当删除发生时，Lucene做的只是将其标志位置为删除，但是文件还是会在它原来的地方，不会发生改变
- 所以对于更新来说，本质上它做的工作是：先删除，然后重新索引（Re-index）

为什么segments不可变?

- 不需要加锁，提升了并发性能
- 查询出的数据就能一直保存在缓存中，比如过滤查询filter就使用了缓存。
- 可以节省cpu和io的开销

02 基本概念、倒排索引、分词、相关性

Elasticsearch节点scale-out



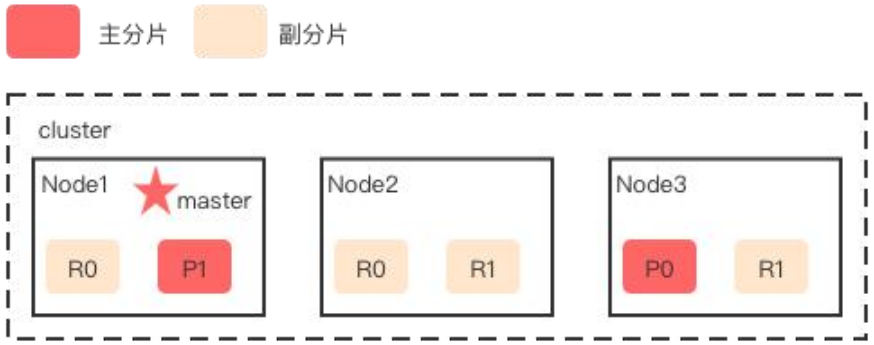
PART 03

文档GET与Search

03 文档GET与Search

文档操作

03 文档GET与Search



分片策略
索引表的allocation主要由allocator与decider来实现，

1 allocator主要负责收集信息，寻找最优的节点来分配分片
源码所在包org.elasticsearch.cluster.routing.allocation.allocator

2 decider主要是做决策是否将分片进行分配
源码所在包 org.elasticsearch.cluster.routing.allocation

index.routing.allocation.total_shards_per_node
默认值不限制，限制单个索引在每个节点上最多允许的分片数目

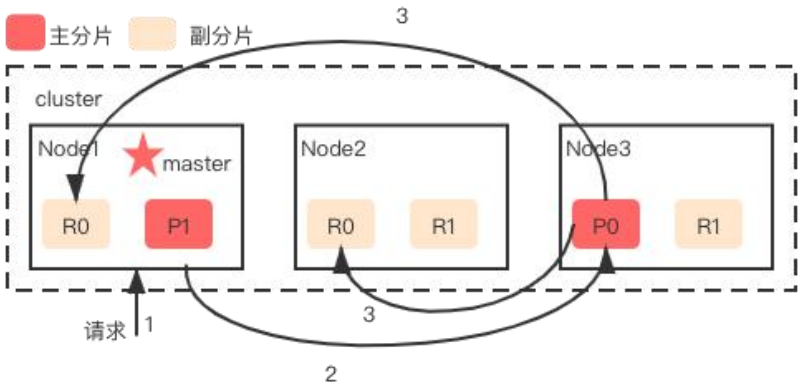
常见的分配策略

DiskThresholdDecider 磁盘空间控制器
FilterAllocationDecider 分配过滤控制器
MaxRetryAllocationDecider 最大尝试次数决策器
ShardsLimitAllocationDecider shard分配的限制控制器

诊断分片为什么没有分配成功

```
POST _cluster/allocation/explain
{
  "index":"homepage_package_index_v2.0.0",
  "shard": 3,
  "primary": false
}
```

03 文档GET与Search



新建，索引和删除文档

1. 客户端向 Node 1 发送新建、索引或者删除请求。
2. 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。
3. 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。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。

03 文档GET与Search

org.elasticsearch.cluster.routing#OperationRouting

calculateScaledShardId方法

计算文档应该落在哪一个分片

hash = Murmur3HashFunction.hash(effectiveRouting) + partitionOffset;

Math.floorMod(hash, indexMetaData.getRoutingNumShards()) / indexMetaData.getRoutingFactor()

effectiveRouting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。 effectiveRouting 通过 hash 函数生成一个数字加上一个偏移位，然后这个数字再与 routingNumShards 取莫得到的余数，最后再除以routingFactor 得到最终结果数 。这个分布在 0 到 number_of_primary_shards-1 之间的数字，就是我们所寻求的文档所在分片的位置。

this.routingFactor = routingNumShards / numberOfShards;
routingNumShards 默认等于numberOfShards

文档的详细写入过程，可参考 Elasticsearch写过程源码浅析

wait_for_active_shards

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

wait_for_active_shards 参数的值可以设为 1（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。
index.write.wait_for_active_shards

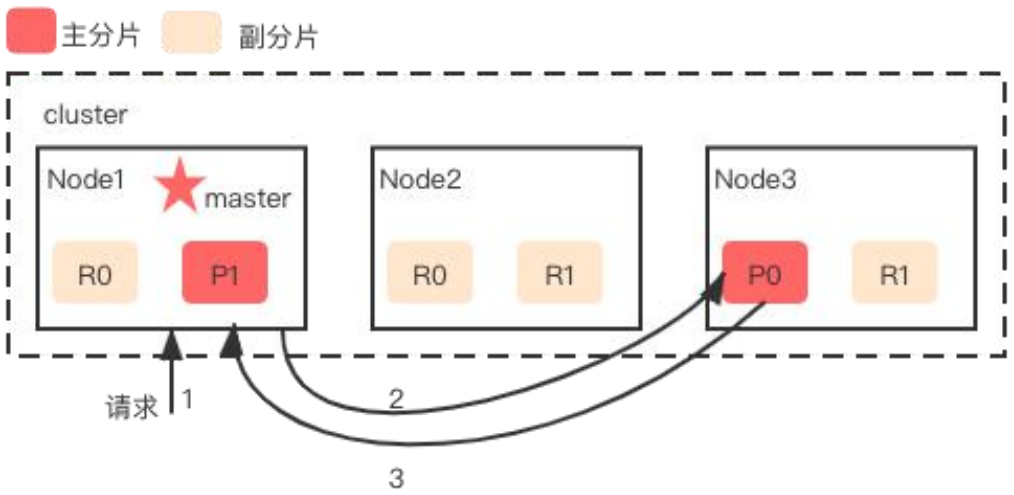
timeout

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： 100 100毫秒，30s 是30秒。

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。

03 文档GET与Search

取回一个文档



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。

官方文档参考

03 文档GET与Search

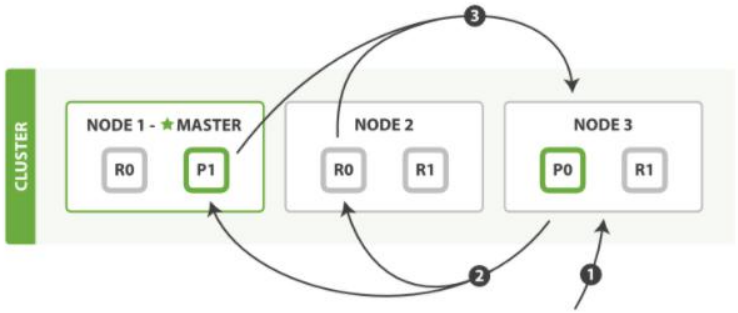
文档操作实战与分析

03 文档GET与Search

Elasticsearch的Search原理

03 文档GET与Search

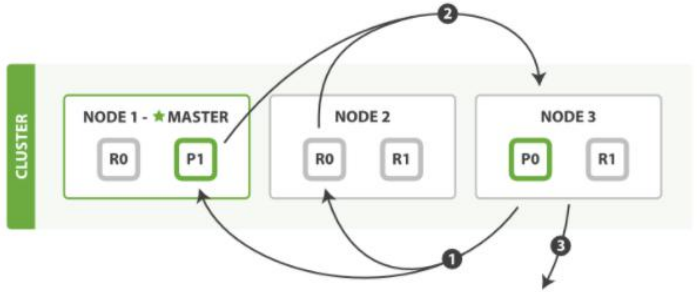
分布式搜索查询阶段



查询阶段包含以下三个步骤:

- 客户端发送一个 search 请求到 Node 3，Node 3 会创建一个大小为 from + size 的空优先队列。
- Node 3 将查询请求转发到索引的每个主分片或副本分片中。每个分片在本地执行查询并添加结果到大小为 from + size 的本地有序优先队列中。
- 每个分片返回各自优先队列中所有文档的 ID 和排序值给协调节点，也就是 Node 3，它合并这些值到自己的优先队列中来产生一个全局排序后的结果列表。

分布式搜索的取回阶段



分布式阶段由以下步骤构成:

- 协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求。
- 每个分片加载并 丰富 文档，如果有需要的话，接着返回文档给协调节点。
- 一旦所有的文档都被取回了，协调节点返回结果给客户端。

为什么要两阶段?

03 文档GET与Search

深分页问题、search_after、scroll

03 文档GET与Search

搜索的类型

query_then_fetch

1. 发送查询到每个shard
2. 找到所有匹配的文档，并使用本地的Term/Document Frequency信息进行打分
3. 对结果构建一个优先队列（排序，标页等）
4. 返回关于结果的元数据到请求节点。注意，实际文档还没有发送，只是分数
5. 来自所有shard的分数合并起来，并在请求节点上进行排序，文档被按照查询要求进行选择
6. 最终，实际文档从他们各自所在的独立的shard上检索出来
7. 结果被返回给用户

dfs_query_then_fetch

1. 预查询每个shard，询问Term和Document frequency
2. 发送查询到每隔shard
3. 找到所有匹配的文档，并使用全局的Term/Document Frequency信息进行打分
4. 对结果构建一个优先队列（排序，标页等）
5. 返回关于结果的元数据到请求节点。注意，实际文档还没有发送，只是分数
6. 来自所有shard的分数合并起来，并在请求节点上进行排序，文档被按照查询要求进行选择
7. 最终，实际文档从他们各自所在的独立的shard上检索出来
8. 结果被返回给用户

PART 04

简单搜索

04 简单搜索实战

轻量搜索

04 简单搜索实战

过滤统计

04 简单搜索实战

精确值查询

04 简单搜索实战

组合过滤器

THANK YOU