搜索引擎及Lucene基础

经典的系统结构 --- 中小型

有一个高可用节点 --- nginx或者ha proxy ---反代http请求

运维 web系统--- 非常关键

Openstack/docker --- 也是一种运维

一个企业用来展示自己风貌 都是通过web界面的

必然 运维主要精力就是涉及到web系统

还有就是游戏服务器运维 ----- 这种游戏运维 用户可能自己的专属协议 没有web界面 分区 3-5台服务器 ---- 用户量大了 水平扩展就行 --- **运维模块化**

Web系统通常是动态 + 静态

并发量大

单台服务器 响应量大概在200上下 （服务器自己的配置 运行的业务复杂度）

----- web服务器配置是一样的

----- 跑一个压力测试 能评估出来

静态内容可以缓存

动态内容也可以缓存 --- 原始状态是动态 执行之后 就是静态的

用户第一次请求 之后 确定没有什么变化 那就缓存一段时间

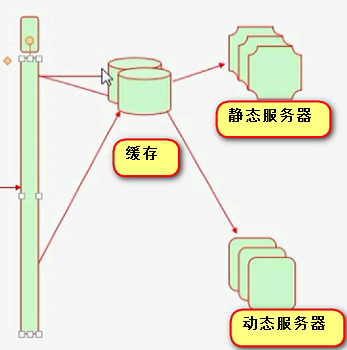
一个页面有10个资源 ----- 不变的为什么不缓存起来？

----------------

动态内容一样可以缓存

缓存哪些 缓存多长时间 需要自己定义缓存规则

但是对于缓存服务器 就是键值服务器



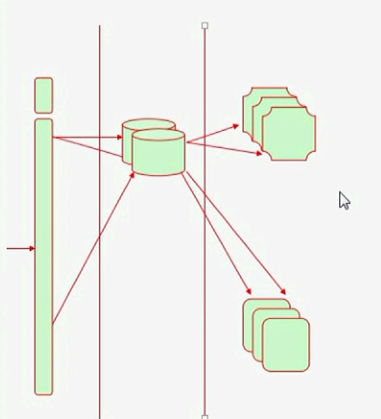
如果90%缓存命中 10%没有命中

----- 访问量有峰值 有低谷 ---- 比如高峰期在午饭期间 --- 下午3-4点 计算这个峰值期间有多少个请求 ------ 保证最快的时候 我们的缓存命中率多高 --- 看看高峰有多少请求到原始静态服务器 和原始动态服务器

------ 现在有一个需求 通过日志记录分析用户的行为，请问 这个日志应该来自于哪？

1. Ha proxy B. vanish C. 后面的动态(LAMP)/静态内容(Nginx)服务器

假设是三层 --- 接入层 缓存层 服务器层



服务器层搜集 ---- 90%请求没有到这里来

缓存层 ---- 没有问题 命中和没有命中都在了

接入层也可以 ---- 很大的站点 前端接入了CDN --- 但是CDN没有命中的时候 还是没有办法记录到

统计一个站点的PV 几乎所有的资源都是动态的 ---- 很多url --- url引入了静态链接

统计PV的时候 可能包含了100个资源 ---- 但是 PV只能算一次

统计PV的时候 一定对访问入口才统计的

如果没有在CDN缓存请求 那么接入层的日志就可以反映

真正的CDN可以挡掉80%的静态文件

UV指的是接入的IP --- 统计独立的不同的IP就可以了

统计PV ---- 10000个资源 整合成100个页面 ---- 大体上哟多少个用户访问入口是可以访问出来的 ---- 直接分析用户访问的url --- 比如jpg等等都去掉

真正访问分析不一定这些 ---投放了这1w件商品 点击量大 有的点击量 ---- 判定哪个商品在哪个季节受欢迎 ==== 或者对商品的喜爱程度有区域性的 ====== 这样就对区域做分析 ----- 这样就可以在前端对不同的区域的用户投放不同的推荐 ---- 这个是推荐系统的产生的 ---- 访问行为大多数就是基于访问日志产生的

==== web站点日志分析 最后形成推荐系统 是非常关键的

==== 所以web站点下面要有日志分析系统

----- 所以因为日志是分布存储在不同的web服务器上面的 但是要统一分析 所以 ---- 需要有一个统一的日志分析存储 ----

\*\*需要日志收集器 --- rsyslog 这个也能搜集日志 启动监听TCP/UDP在某个端口上 就能搜集日志

但是 http https 根本不是rsyslog搜集 那就不行了 --- 其他机制 ---- 写一个程序 在每一个机器监听对应的文件 --- 然后把对应的文件读取到目标的同一日志存储就可以了 这个在较小规模没有问题 --- 如果应用规模非常大

前端百十来台缓存服务器 --- 就有可能每秒钟产生的日志量上w条 ----- 近万条日志信息 这样统一的存储（单台）无法承载 --- 还要再存储上作分析 对IO来讲 又是压力 ---- 每次分析的又不一样 ---- 较大规模站点 搜集产生很大问题

假设加入每秒钟日志w条 --- 一天就是8亿条记录 ---- 8亿条记录 什么概念？ 每一条信息combined格式 --- 1000w条大概1-2G的数据 ----- 8亿条数据量有多大 --- 一天日志量40G 一周呢

现在要在这么多的数据量 看看某一个PV UV 等等 更何况频繁程度复杂的

如果日志分析是针对不同的部门 多个商家 ---- 日志分析不仅站点需要 加入没有第三方卖家 那还简单点 如果有第三方卖家 对方又要看看 这样麻烦大了 别人可能一小时打不开 然后不在你这做了 ==== 这样日志的存储 分析 对IO挑战非常大

----- 单个节点不行 然后就是分布式

MySQL支持流式化数据 Mongo等等 存储系统 比文件系统应用场景不同

日志可以存储到mongo中 ---- 一种和mySQL差不多 但是mongo本身就是分布式能力 --- 有三个节点 就会天生给你分片 ---- shard ---- mongo能存储是没有问题的 但是 分析怎么办？

关系型数据库本身并发能力都非常有限 ---- 高并发场景 1w/s插入到mysql 然后又查询 ---- mysql就完蛋了 ---- 复杂的内部约束 应付不了高并发场景

日志文件数据是半结构化的 很难完整插入到mySQL

MySQL查询 要是模糊查询 当你的模糊查询最左侧是通配符 无法使用 一旦面对这种场景 就完蛋了

所以 MySQL对这种场景出现时 无法实现随意的灵活性查询 --- 并且对索引依赖很大

Mongo也是差不多 这是因为自身带的查询接口的能力有限 ---- 这时候就需要搜索引擎完成这个功能

--- 也就是任何一个客户输入一个关键字 就几乎进行全文搜索



MySQL中 左侧通配符 就无法使用索引

但是 8亿条记录 没有索引 你看什么概念 所以MySQL不适用

---- 需要搜索引擎 完成全文搜索 并且 根据任何关键字搜索 并且 作为智能化的智能匹配

---- 搜索电脑或者笔记本 ---- 电脑 或者 纸质笔记本

MySQL 这些是不可能的

----- 这种工具就是搜索引擎 是一个**系统自身应有的功能**

程序是算法+数据结构 --- MySQL存储结构 上面放的就是SQL引擎 关系型的模型

但是全文搜索算法 底层就是支持对应的算法

每一个系统都是两部分组成：内部的存储结构 + 前端的访问接口 ---- 有的只提供了存储 前端访问接口 但是不负责处理引擎

我们这里的搜索引擎：存下来+能够对存下来的数据进行模式分析 根据用户指定的filter过滤出用户的搜索

海量日志分析就需要有搜索引擎

大web站点 都需要搜索引擎

搜索引擎 –一般两个部分组成

---- 索引链（数据要能存下来 并构建索引）（没有索引 搜索任何工具在上百亿条符合你的数据 一定非常慢 ----- MySQL数据量较大 也要索引）==== 这段叫存储段

---- 第二段 叫搜索引擎段 也叫搜索组件

每一个搜索引擎大部分都是这两部分组成 ---- 也就是称作为程序组件部分

 ---- 用户通过界面键入查询的关键词 然后通过搜索组件对用户输入的次进行分析 进行正规化 ---- 正规化（键入的是NoteBook ---- 问小写的notebook 近义词要不要搜索 这些搜索组件要构建出来）

二者之间的结合点就是索引

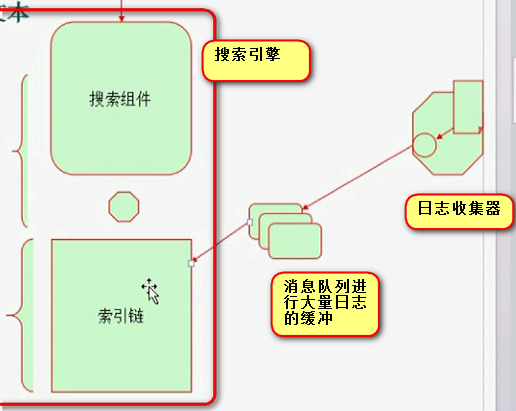
所以 索引链的工作就是把数据搜集起来 构建索引

---- 问题：数据从哪来？---- 必须有位置把数据搜集过来？

如何搜集？一种pull 另一种push

\*\*自己的爬虫程序 就是一个浏览器 不需要展示出来 自己访问一个url –顺着这个url藤 分析url对应的url 爬遍全球的所有页面 --- 搜集完成之后 才能让用户支持关键字搜索 --- pull 是搜索引擎自己拉取数据

\*\*日志搜集 ---- 日志信息都放入一个文件中 **访问位置是固定的** 不像互联网盘根错节----- 所以 不用爬虫随便 --- 只需要对固定的位置做简单的搜集就可以 ---这时候让web服务器装agent 监控日志文件 --- 一旦这个agent监控的文件有新的数据进来了 就发送一个数据流给搜索引擎的索引链 ==== 缺点 几百台服务器 建立几百个并发连接接受别人的数据量 如果每一个数据量很大 那么我的网络IO无法应付 ---- 这时候前端做一个缓冲 ---- 就是一个队列 --- 我们服务器**按照自己最大的能力**就从队列上取数据 --- 然后在本地完成索引构建 --- 索引链必须要完成索引数据的构建的



搜集过来之后 想办法检索原始内容（易于构建索引的数据格式 数据结构） --- 根据原始内容创建document –对文档document完成索引构建

ELK的存储格式 是 文档格式 --- document

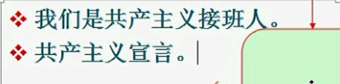
之前的mongo就是一个document 每一个文档类似于MySQL表手中的一行 – mySQL每一行都是有固定结构的 ---- 但是nosql没有这个要求 ----- 文档一个表中 文档的结构都是不一样的 --- 存储了大量的键值 ---- 可以嵌套 --- yaml格式 也存储了大量的键值

非常灵活的结构 --- 存储数据的表现 表现形式是文档 不是行 ==== 对文档中的数据创建索引 ---- key叫什么名字 值要切词 等

比如有一句话 我们是共产主义接班人 --- 切词 ---- 我们 是 共产主义 接班人 --- 以后搜索接班呢？ 这就搜索不出来了

切词 对于不同的场景 很多种不同的 --- 不切词 才能做搜索 === 如果不切词 就只能做模糊查询了 ---- 并且模糊查询还是全文搜索的

不仅仅是切词

还有一个是  搜索共产主义 哪个更接近我们的需要？ 谁拿的钱多？ 谁出钱 权重就调整大一点 === 每个文档都由权重 --- 权重从哪里来？

\*\*哪个文档短 权重更大

\*\*更简单的 谁给的钱多 就用谁的

----- 也就是同一个词 有很多自己的计算方式

比如google的pagerank ---- 同样共产主义 如果一个文档的有效引用次数非常多 点击数非常高 那么 就分数高 ---- 计算起来很麻烦

搜索组件 --- 为用户提供一个界面 进行搜索操作

搜索组件主要提供用户页面的

===== 汇总一下

搜索引擎 最前端的是用户 --- 用户的需求必须通过用户访问接口UI进来 ---- 要不要对用户键入的数据构建出可用的查询呢？

 要不要切开？ --切割开

 只要出现其中的一个 就行

提交的时候 会有html标签 ----- 服务器要把这些标签去掉

----- 就是想方设法构建成一个可以构建的查询

构件查询：大小写转换好 切词 ---- 构件查询

下面就是运行查询

 查询都在索引上执行 ---- 通过索引完成查询操作

索引就是索引链构建的 ---- 这样就达到索引链了

上面就是搜索组件

如果索引返回内容通过用户接口显现出来

运行查询 就要返回结果

 大括号括起来的就是搜索组件

---- 抓取到原始内容 -> 原始内容抽取 作分析 构建成文档格式 -> 文档分析 （切词）-> 构建**倒排**索引

SQL索引就是正排的 --- 正排索引

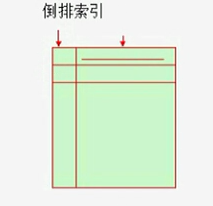
搜索引擎的索引叫倒排索引 ----

任何文档获取之后 都应该切词 --- **每一个词出现在哪些文档中**

----- 根据词来找文档 -----但是MySQL就是根据记录来找词

两种不同的搜索机制

因此 这个索引中 每一行就是两个字段：每一个被切出来的词 后面是这个词所在的文档的编号



就搜索我们 共产主义 两个词

找我们 1

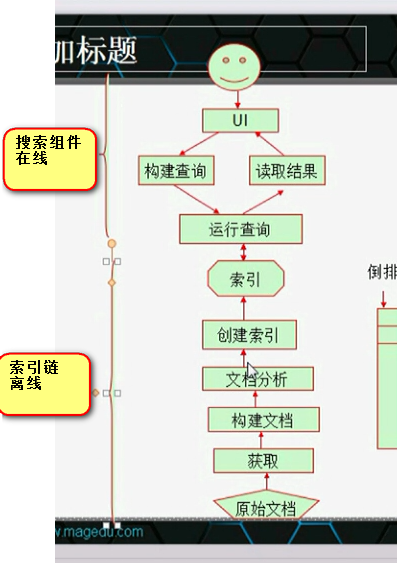
共产主义 1 2

所以 第一个是部分匹配 第二个是精确匹配

这种就是倒排索引机制

理解 如何切词 + 如何排序算法（比如引用次数 让几万个站长把你的网站链接上）

索引的构建就是这样的顺序

【原始内容 还不是文档格式 需要变成文档格式】

有了索引 可供搜索

二进制怎么构建文档？很麻烦 后来可以搜索图片 是根据图片标题 （实际是tag）

每一个文档 无非就是键值构成 ---- 分析过程涉及到非文本数据的分析 –文本数据 如果是html文档 还要把html变成纯文本

---- 必须把文档进行切词 每一个切出来的就是Token ---- 哪些是词？ ----- 不同的方式 不同的语言都不一样的切词

---- 拼音拼写错误 是否要给词语加上同义词 大小写不匹配 需要一些词干分析器

---- 开源程序中 能完成这些的 最著名的一个就是Lucene

Lucene只是一个索引链 没有提供任何索引组件 ----- 可以把原始内容分析 构建成文档 ---可以对文档进行分析 --- 并构建出可供查询的索引 ----

===== Lucene 有两个问题

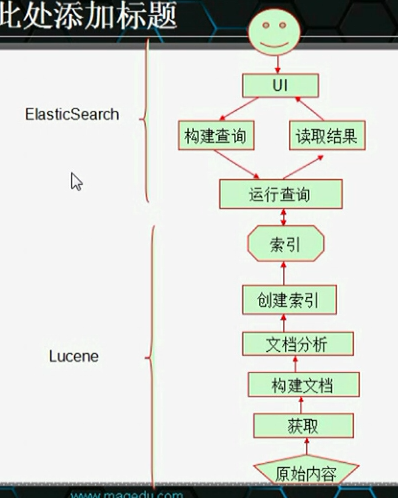
(1). Lucene自己不获取文档

(2). Lucene自己不提供搜索界面

所以 站在这个角度 Lucene只是一个库 ----- 那么搜索组件的 目前就是ElasticSearch---- 任何一个公司 有程序员 就可以利用Lucene写程序 ---- 后来出现了公共的ES

=====

**ES仅仅是搜索引擎的一部分 核心是Lucene**



对搜索组件 要提供UI 并且建立查询 ---- 也需要对提交的标签进行切词 等等各种复杂的运算 --- 根据用户提交的请求 转换成可查询的语句 ---背后要做一些很复杂的boolean运算 ---- 这些都是所谓建立查询负责的

建立完查询之后 就运行查询

给出的文档是根据score排好序的 ----- 搜索有各种模型

搜索引擎中的后台索引构建组件

======= Lucene

\*\*文档 document

**文档是搜索的原子单位 --- 包含了一个或者多个域field的容器** Field ---- document是由filed和value组成

一个document可以包含一个kv或者多个kv

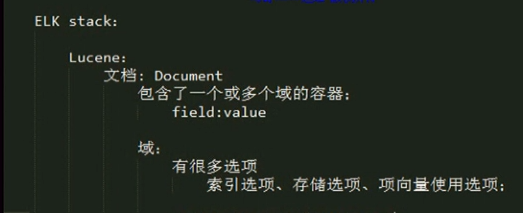
**真正搜索 记住 要对value搜索**

---- Lucene提供了类似于传统关系型数据库的存储和查询能力 但是没有确定的全局模式

---- 但是Lucene没有全局Schema 每一个文档都是自由定义的 【数据库的schema就是 每一个数据都有自己的所在的表 表中的字段是提前构建好的】

\*\*域 创建域的时候 可以为域指定多个选项来控制lucene 将文档添加域索引后 可以执行哪些操作 ===== 这个就是**域的分析过程**

域有很多选项 其中包括索引选项（定义如何可以被索引） 存储选项（定义如何被存储） 向量使用选项 ---- 这三个选项可以组合 也可以单独使用



索引选项主用用于通过倒排索引 来控制文本可被搜索

**\*\*就是说 每一个文档只有成为索引中的项 才可以被搜索**

比如 this is a desk. ---- 其中有意义的词是desk ----- 所有可被搜索对象一定是倒排索引中加载的词

索引选项用于通过倒排索引是否可以被搜索

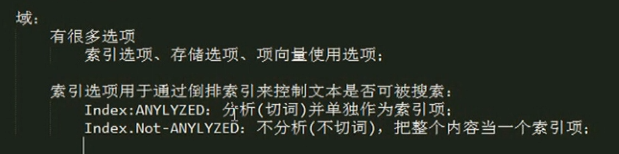
\*\*Index.ANALYZED

如果一个索引 定义叫ANALYZED ----- 表示分析 ----- 表示要使用分析器 对域内的词进行切词 ---- 这时候 域中的词被切词 然后放到倒排索引中 这样域中的词就能够被搜索了

\*\*Index.Not-ANALYZED

对域**做索引 但是不做分析** ---- 就是整体域不进行切词 作为一个词出现出现在倒排表中

【**不分析就是不切词的意思**】

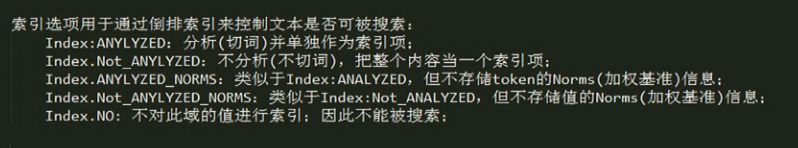


对于Index.ANALYZED 可以搜索到域中的单独的每一个词 但是 对于Index.Not-ANALYZED就只能对整个域的值进行搜索 不能对其中的词进行搜索 不支持部分搜索

\*\*Index.ANALYZED\_NORMS: 是上面Index.ANALYZED的变体 ---- 每一个文档是有权值的 ---- 切词 但是不存储TOKEN的Norms（Norms叫加权基准信息）

\*\*Index.NOT\_ANALYZED\_NORMS 类似于Not\_ANALYZED 并且不存储Norms

\*\*Index.NO ---不对此域做索引 这样这个域不能被搜索



以上叫索引选项

---- 下面是存储选项 主要用于确定是否需要存储域的真实值 ---- 如果用户搜集来的数据 叫做title: “this is a Notebook” ----- 切词之后 分析法则 Notebook转换成notebook ---那么搜索出来的时候 是大写还小写？这就是存储域内分析前的真实值

store.YES ---- 存储真实值 --- 意味着原始字符串也要保存在索隐中 因此要占用额外空间 – 一个数据要存两遍

store.NO ---- 不存储真实值 ----没法展示真实值

store.NO和Index.ANAYLYZED经常结合把一个大的Web页面的正文进行切词 分词之后不保存原始值

这叫存储选项

\*\*域向量或者项向量选项

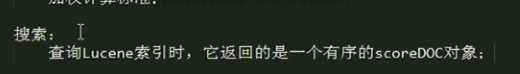
用于域在搜索期间 该文档所有的唯一项都能完全从文档域中检索使用 ---- 用处不大

文档和域还有加权操作 ---- 默认权值一样 ----- 改变某一个文档的加权因子 ----加权因子就是广告费 --- 加权计算标准有很多方案

Google使用PageRank来进行加权操作

如何执行搜索？

Lucene如何在自己的索引上面进行搜索？

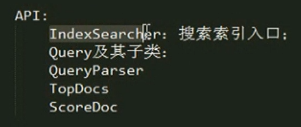


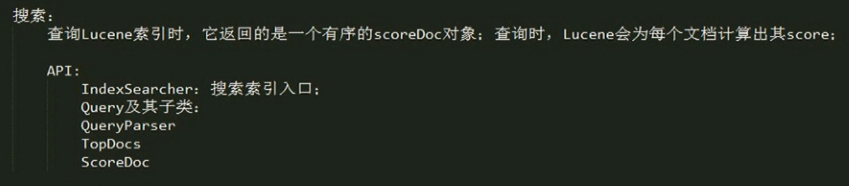
ScoreDOC 有序的 --- 根据加权标准 为每一个文档计算出齐Score ---根据分值进行排序

ScoreDOC对象自身并不会进行文档匹配 ---- 需要ID查找原始文档

Lucene通过大量API完成查找

TopDOCs: Lucene每次返回结果的ScoreDOC对象会放在一个称为TopDocs的数组中 ---- 用于保存一次查询分数最高的前10个ScoreDOC





那么Lucene中 通过IndexSearcher +Query(及其子类) 来构建build一个查询 然后 通过QueryParser完成Query分析 然后执行搜索 ---- 搜索结果通过ScoreDOC返回 封装在TopDocs

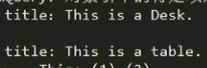
---------------

Lucene可以支持非常多元化的查询

Lucene最终都是通过调用IndexSearcher的search方法完成搜索 ---- 但是 要把Query实例传递给search方法 ---- Query很多种 有这样几个

TermQuery ----对索引中的**特定项**进行搜索 --- 倒排索引就两个字段 一个就是项 另一个就是项出现在文档的编号 ----- Term就是**最小的索引片段** 每个Term包含一个域名和一个文本值

比如 有两个文档



倒排索引如下

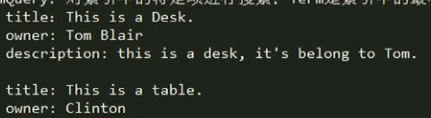


搜索时候 执行TermQuery ---- 搜索title这个域（注意 搜索某个域 前面说过是搜索这个域的值 那么 搜索title这个域 指的是搜索这个title后面跟的值 ）

---- 搜索title这个域中出现table 只有第二个文档满足要求

---- 搜索title这个域中出现this 第一个和第二个文档都满足要求

现在每一个文档未必有一个键值对



这样 你如果要对这两个文档建立的倒排表进行索引搜索desk的话 可能出现在title 也可能出现在description

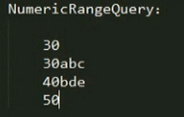
但是 使用了TermQuery之后 指定在哪个域中进行搜索

\*\*TermRangeQuery

就是 索引中每个Term中的值 都会按照字典序 进行排序 然后允许在Lucene的TermRangeQuery中的范围进行搜索

TermRange表示有多个域 ---- 说白了就是 能搜索指定的多个域 不是单个域

NumericRangeQuery 搜索会搜年龄或者字符串



搜索30-50的字符串 这个都对 但是搜索30-50的数字 第一个和最后一个符合

PrefixQuery:用来搜索指定字符串开头的项

比如搜索以this开头的项 ----- 类似关系型数据库的查询

BooleanQuery是用来实现组合查询或的 ---- 比如年龄>20并且姓名tom开头 ---boolean Query连接

PhraseQuery 能够位置信息定位对应文档 了解就好

WildCardQuery 通配符查询 --- 可以结合问号或者\*查询

FuzzyQuery模糊查询 用于匹配相似的项 ---概率计算 用到Levenshtein完成索引项中域搜索条件的匹配程度 Levenshtein是距离算法

