elasticsearch

背景

互联网主机数

互联网基础设施：个人设备普及，网络，协议，

互联网人数：40 亿

搜索引擎变化

1. 主动发现
2. 被动接收，个性化

人与互联网交互的入口，流量入口

发展历程

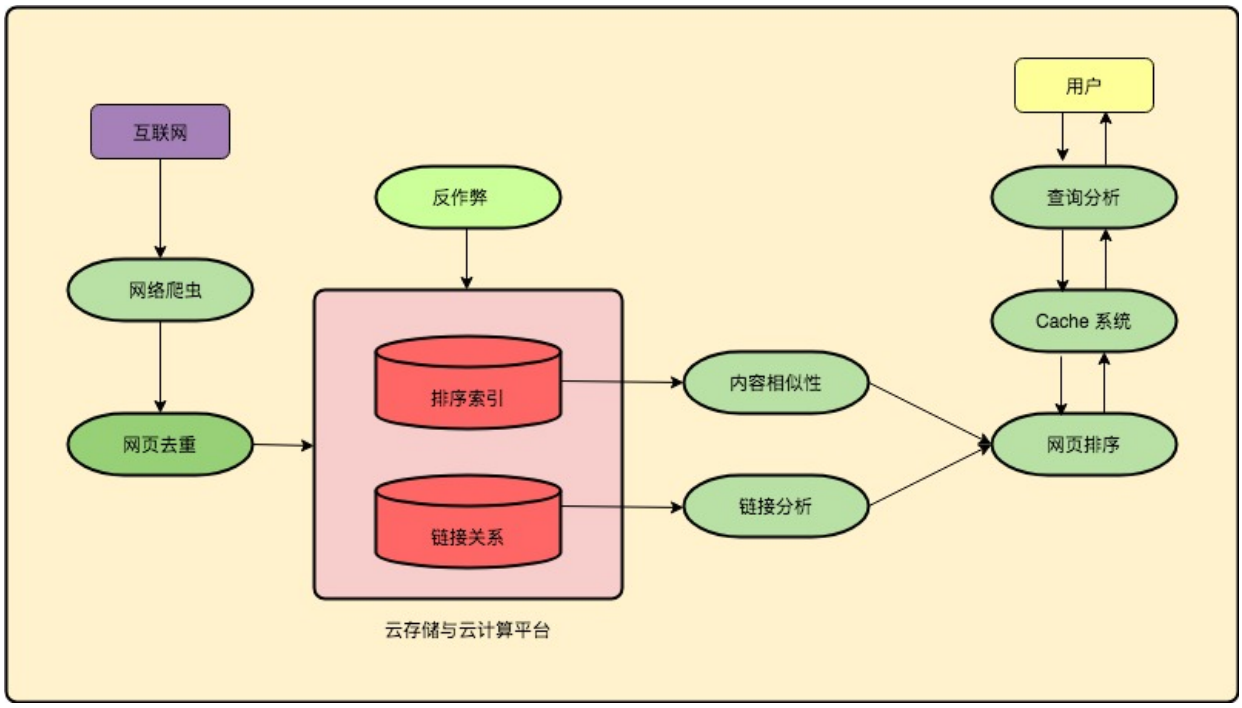
- 分类目录的一代：人工整理分类目录，如 Yahoo
- 文本检索的一代：经典的信息检索模型（布尔模型，向量空间模型，概率模型），如 Alta Vista
- 链接分析的一代：分析网页中的链接。如 Google 的 PageRank
- 用户为中心的一代：以用户需求为中心的个性化搜索。以用户位置以及历史信息等为依据

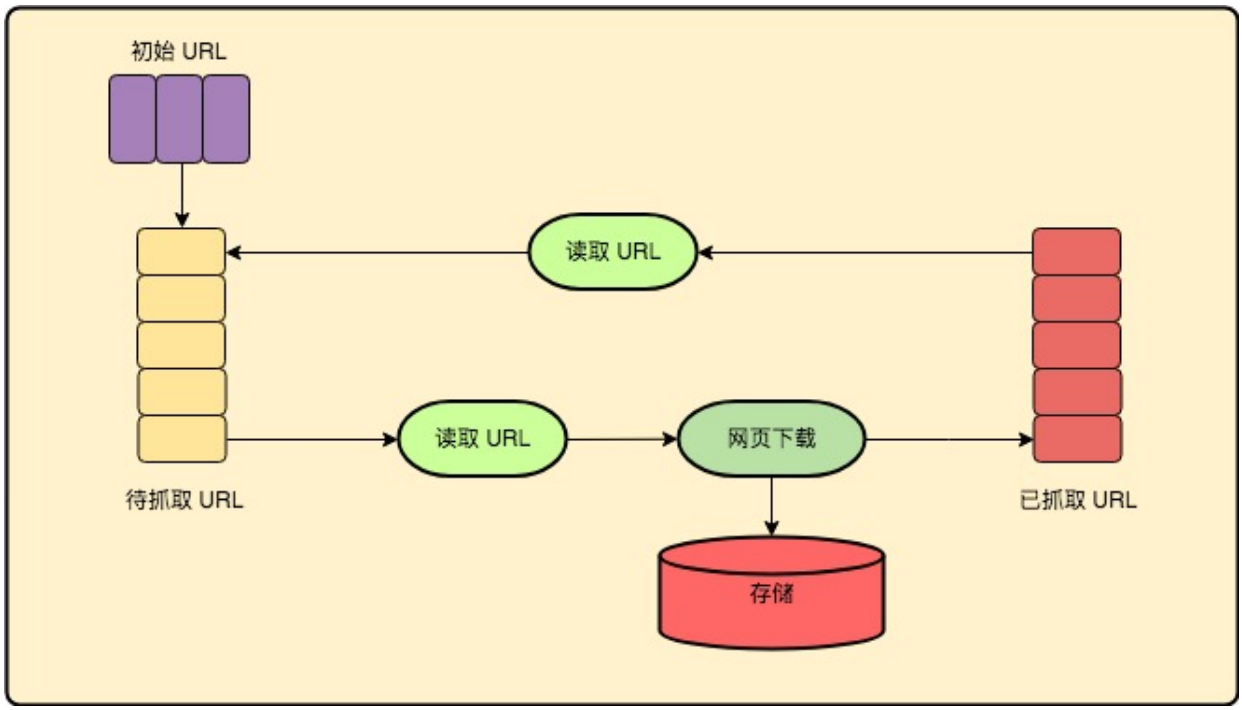
核心目标

更全：网页数量。爬虫，存储等

更快：查询速度。依赖爬虫，存储，索引，缓存等

更准：核心竞争力。通过用户研究，相关性分析（网页去重，排序），信息质量（反作弊）等





分布式爬虫程序 + 分布式存储

索引过程概述：

1. 从未爬取 URL 队列，取一个 URL，爬取网页，网页入库；解析关联 URL，与已爬取 URL 队列查询，如果已经爬过，略过，没有爬过，将其加入未爬取 URL 队列，继续步骤 1
2. 另外对入库的网页进行索引分析，建立缓存，去重，反作弊等等

查询过程

1. 先查缓存，如果缓冲命中直接返回，如缓存没有命中，继续步骤2
2. 词法分析，语法分析，倒排索引查询关键词，更加内容相似度和链接分析对网页进行排序，返回 top K。

爬虫

分类

互联网页面分为

可知网页集合

- 已下载网页集合
- 已过期网页集合
- 待下载网页集合

不可知网页集合：占比很高。

优秀爬虫系统的特性

从系统设计角度

1. 高性能：单位时间爬取的网页数量
2. 可扩展性：线性增加节点
3. 健壮性：自身原因（内存满，不明宕机），外部原因（死机，不能解析，安全）
4. 友好性：禁爬虫协议，减少网站负载

从用户角度

1. 网页覆盖率：覆盖率越高越好
2. 时新性：网页的更新越及时越好
3. 网页重要性：网页重要性越高越好

比如 google 有两套系统：Fresh Bot 以秒为单位；Deep Crawl Bot 以天为单位，满足不同频率更新的网页。

网页爬取策略

待爬取网页策略：核心就是越重要的网页越早爬取，因此对网页的重要性评估就是一个关键，比如 Google 就是因其 PageRank 而名声大噪。

1. 宽度优先遍历策略

2. 非完全 PageRank 策略：按批次进行 PageRank。褒贬不一
3. OCIP(Online Page Importance Computation)：类似 PageRank，略优于宽度优先遍历算法
4. 大站优先策略：略优于宽度优先遍历算法

网页更新策略

1. 历史参考策略：泊松分布，去掉广告栏，导航栏
2. 用户体验策略：计算历史版本的重要性
3. 聚类抽取策略：同类的网页更新周期一致，不同类别的网页采用不同的更新频率。好于前两种

暗网爬虫（deep web crawling）

垂直网站

1. 查询组合：富含信息查询模板(Information query template)，ISIT 算法
2. 文本框填写问题：人工启发式加递归

分布式爬虫

主从分布式爬虫：URL 服务器进行 URL 分发，URL 服务器成为瓶颈

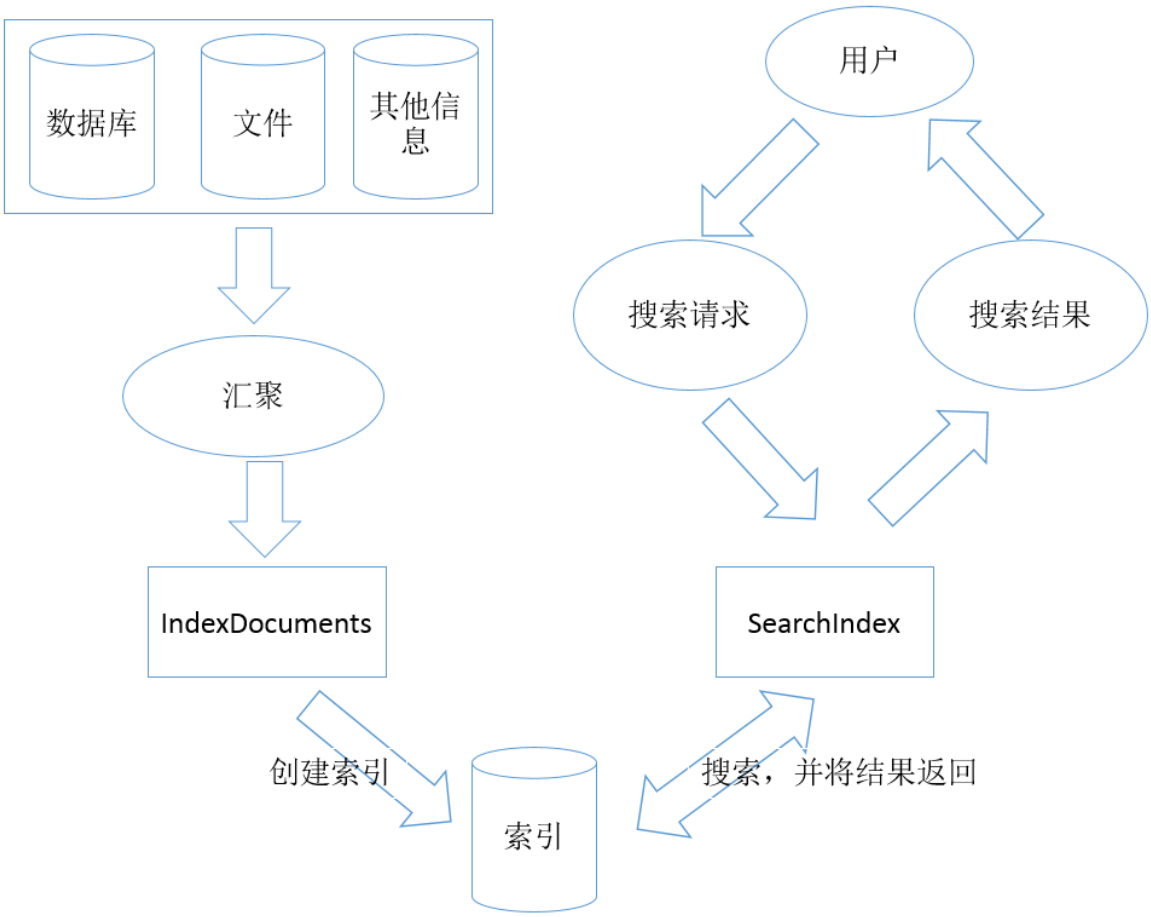
对等分布式爬虫：根据域名一致性哈希计算

索引

数据分类：

- 结构化数据：指具有固定格式或有限长度的数据，如数据库，元数据等。
- 非结构化数据：指不定长或无固定格式的数据，如邮件，word文档等。
- 半结构化数据：如XML，HTML等

搜索结构化数据：对数据库的搜索，用 SQL 语句。再如对元数据的搜索，如利用 windows 搜索对文件名，类型，修改时间进行搜索等。



如何搜索非结构化数据?

传统方法 - 顺序扫描法

比如要找内容包含某一个字符串的文件，就是一个文档一个文档的看，对于每一个文档，从头看到尾，如果此文档包含此字符串，则此文档为我们要找的文件，接着看下一个文件，直到扫描完所有的文件。比如 window，mac 的文件搜索(windows 支持建立索引来加快查询)，linux 的 grep。对于 10 G 的文件估计至少得 1 分钟。如果是一个 1 T 的问题件呢？1000 分钟不止，这远远不能满足可用。

如何解决大数据量的搜索

传统方法存在的问题：如果根据文件名查询非常快，但是根据文件内容查询就非常慢（需要遍历所有文件），如果事先对文件内容建立索引，查询就会非常快。

此外，数据库之所以能搜索的那么快的原因是因为有索引，有时候为了加快搜索，手动建立索引。

所以问题转变为是否可以对非结构化的内容建立索引，具体可以分解为：

1. 索引里面保存什么?
2. 如何建立索引?

3. 如何查询?

这便是搜索引擎索引要解决的三个问题。

WAL

索引里面保存什么

单词-文档矩阵

	文档1	文档2	文档3	文档4
单词1	有	有		有
单词2		有	有	
单词3	有	有	有	
单词4				有

实现上述概念模型的方式

1. 倒排索引
2. 签名文件
3. 后缀数

实验表明：倒排索引是实现单词-文档映射关系的最佳方式。

基本概念

文档：以文本形式存在的存储对象，可以是各种文档格式。

文档集合：若干个文档的集合

文档编号：搜索引擎内部，唯一表征一个文档的编号

单词编号：搜索引擎内部，唯一表征一个单词的编号

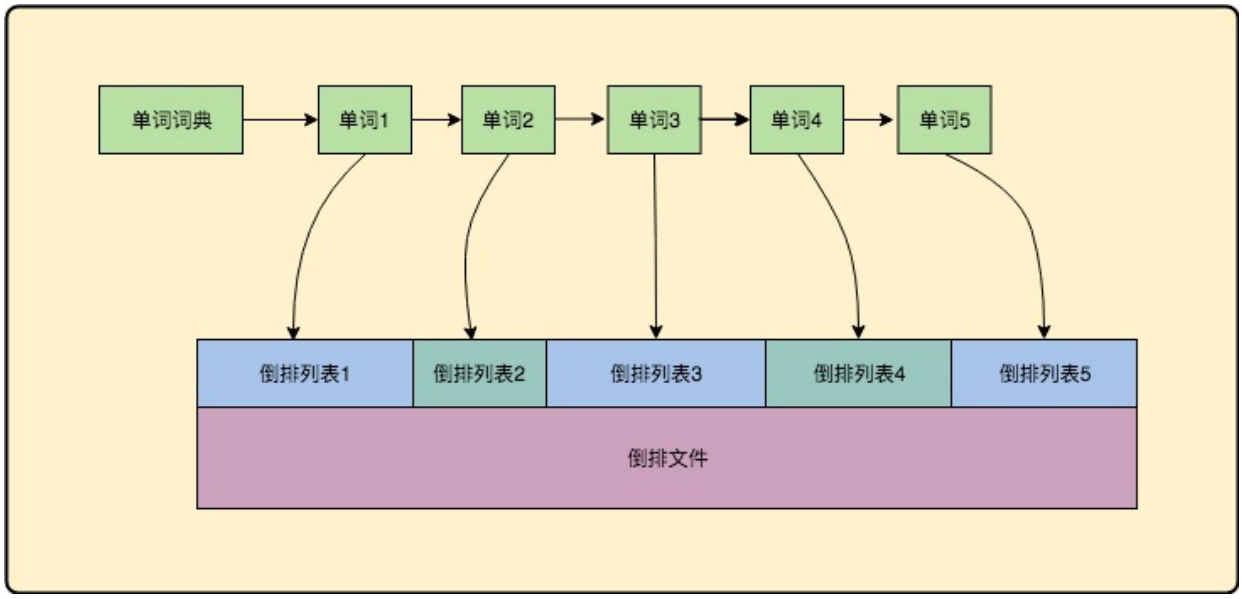
倒排索引：由单词词典和单词对应的倒排列表组成。

单词词典：所有文档的单词集合，一般用最小完美哈希、双数组Trie 树，M 树（B 树或B+树）保存。

倒排列表：包含某个单词的文档列表及 该单词在文档中的位置，每一项为倒排索引项。由文档名为索引变为文档内容关键词为索引，故名倒排索引。

倒排文件：所有单词的倒排列表顺序存储在磁盘文件，这个文件即为倒排文件

倒排表(Posting List)：每个关键词对应的文档链表为倒排表。



如何建立索引

静态索引

1. 两遍文档法（2-Pass In-Memory Inversion）：内存中建立索引，两遍扫描文档，实际很少用。
2. 排序法：单词 ID，文档 ID 排序生成中间结果，合并中间结果。词典在内存中，三元组可以在磁盘。
3. 合并法：对排序法的优化。完整的内存索引结构。词典也可以在磁盘。

动态索引

已删除文档列表：已删除文档 ID 列表

临时索引：内存实时建立的索引，单词词典和倒排列表存储在内存

倒排索引：初始文档的倒排索引，单词词典存储在内存，倒排列表保存在磁盘

索引建立过程：

新增文档，实时解析，加入临时索引。

删除文档，将文档加入删除列表

修改文档：先删除，后新增。

文档查询过程：

将倒排索引和临时索引合并，并用已删除文档列表过滤。

索引更新策略

临时索引用完时，如何对索引进行更新。

1. 完全重建策略（Complete Re-Build）：当内存达到阈值，对老文档和新文档重新建立索引，将删除老的索引。适合小的文档。
2. 再合并策略（Re-Merge）：当内存达到阈值，将临时索引与倒排索引归并生成新的索引。对于没有变化的老索引仍然需要磁盘拷贝。
3. 原地更新策略（In-Place）：将临时索引追加到倒排索引，初期倒排索引会预留给空间给临时索引，如果空间还是不够需要迁移。出发点很好，但事实上比再合并策略差。
4. 混合策略（hybrid）：高频单词用原地更新策略，低频率单词用再合并策略。

注意区别，完全重建合并的是文档，而再合并合并的是临时索引和倒排索引。

索引建立的完整例子见附录

如何查询

由**文档名到文档内容的索引映射关系**变为**文档内容关键词到文档名**的映射关系，搜索关键词，找到关键词对应的文档，遍历文档，即可找到对应的字符串，这样可以极大降低搜索的文件的数量。

但这样存在一个问题：

1. 几乎所有可以搜索关键词建立索引，空间成指数级上升
2. 索引建立时间

对于问题 1，这就是 trade off，空间换时间的典型案例。

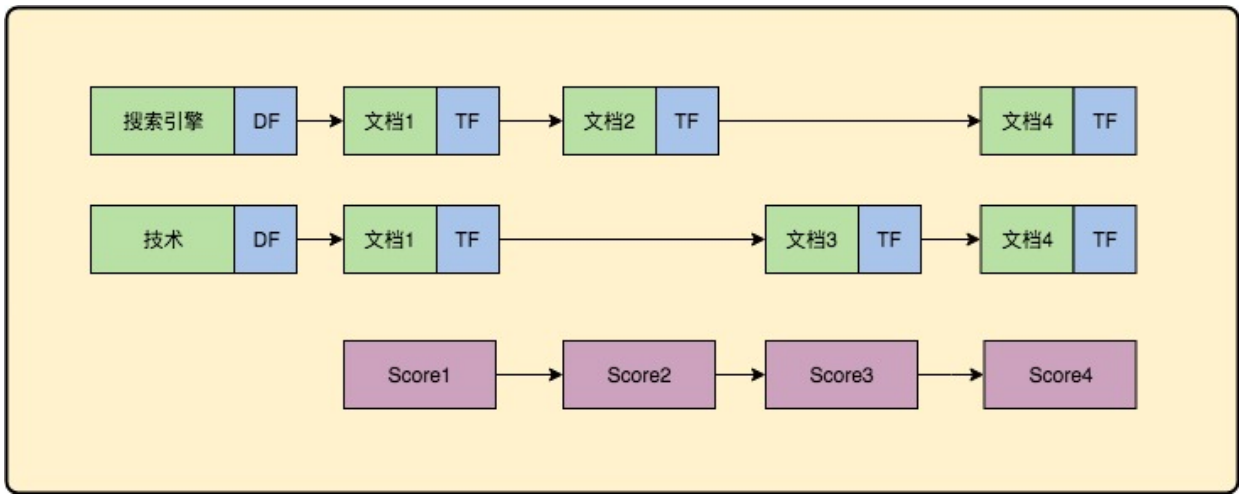
对于问题 2，由于是建立索引需要一次，而查询是多次，即**一次索引，多次使用**，因此可以加快查询速度。

索引方式

实际查询的时候，可能是单词，多个单词，短语，如何建立它们的索引?

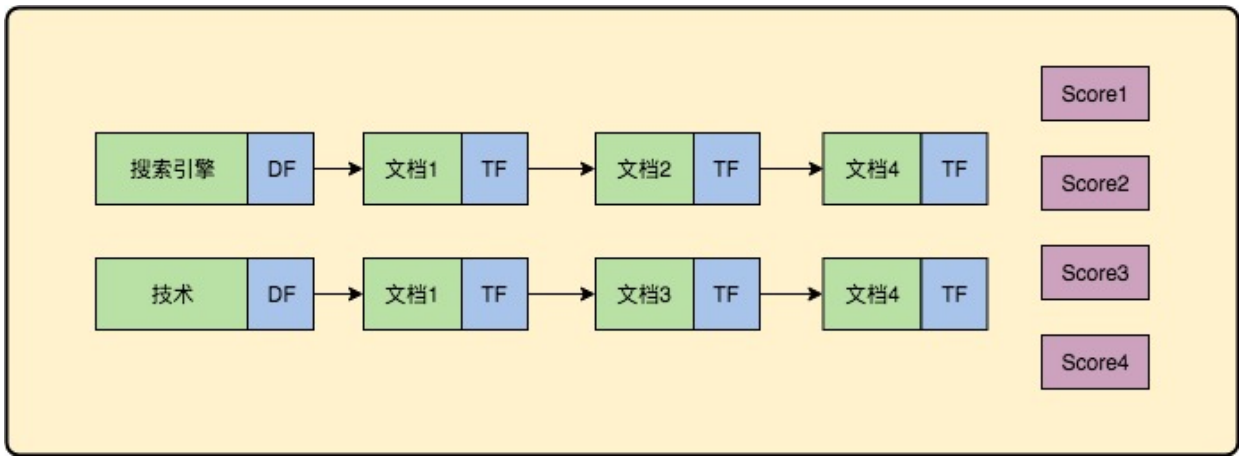
单字段索引

一次一文档：先文档后单词



根据待搜索的单词从内存读取单词的倒排列表，依次对每个文档与搜索单词的关联度计算分数，如果多个文档出现同一个单词，该单词与文档的分数叠加。一般用根堆直接保存 top K。

一次一单词：先单词后文档



根据待搜索的单词从内存读取单词的倒排列表，依次对每个单词与所有关联文档计算分数，如果多个文档出现同一个单词，该单词与文档的分数叠加。一般用哈希表保存所有单词，之后排序取 top K。

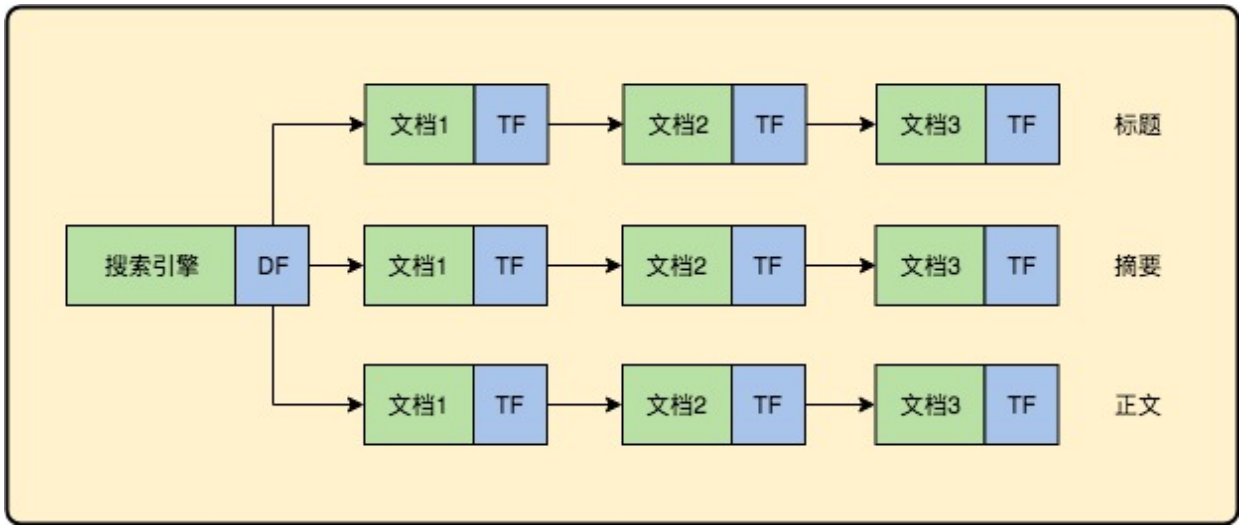
跳跃表

如果倒排表保存的之间是文档 ID，依次一次一文档很高效。但是，实际上，会对倒排索引中的文档 ID 进行压缩，比如，保存相邻文档 ID 的差值，这个时候用一次一文档需要解压所有，并依次比较。通过跳跃表，1. 只需要解压部分数据；2. 无需比较任意两个文档 ID。

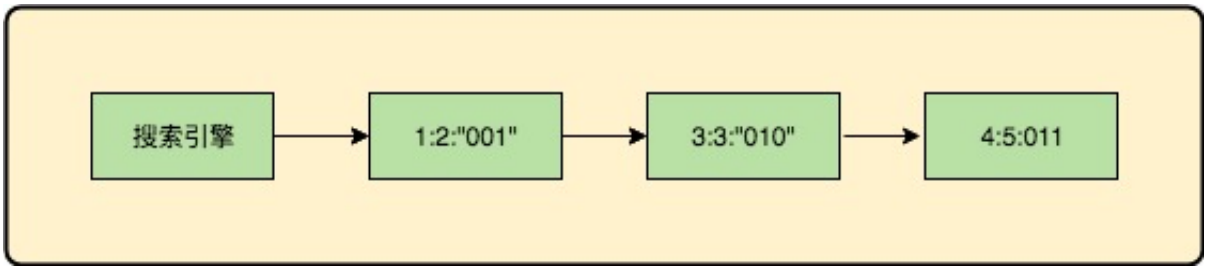
多字段索引

关键词出现在文档的不同位置，权重应该不一样。比如出现在标题与出现在正文的权重是不一样的。

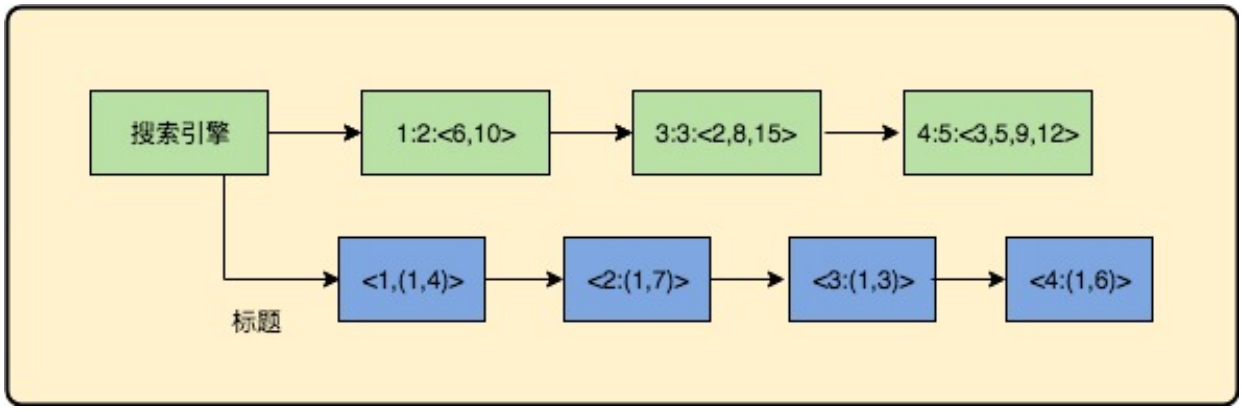
多索引方式：对同一文档的不同字段（标题，摘要，正文）建立不同的索引



倒排列表方式：在倒排列表的倒排项中增加一个字段表明是否出现在某个字段，比如 xxx 分别表示标题、摘要、正文，001 表示只出现在正文



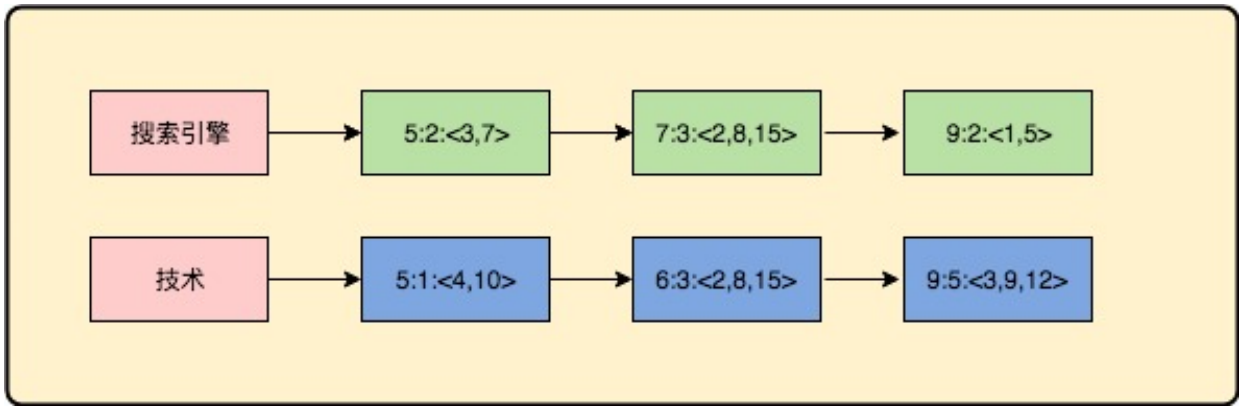
扩展列表方式：用额外的列表记录，实际中用得比较多。



1. 标题列表记录每个文档的标题的索引范围
2. 倒排列表增加字段，记录每个单词在文档中的索引

短语查询

位置信息索引：在倒排列表的倒排项中增加一个字段记录单词的索引信息



如图，搜索引擎出现文档 5 的位置 3，技术出现在文档 5 的位置 4， 说明在文档 5 出现了搜索引擎技术这个短语。缺点是这种方式付出的存储和计算代价很高。适合于计算代价较小的短语。

双词索引：基于短语中两个词最多的实践。首词、下词。适用于对计算代价高的单词。对包含“我”，“的”等停词的短语使用

短语索引：对短语建立索引，适用于热门短语。其实类别与缓存。

实际的查询顺序是：短语索引 -> 双词索引 -> 位置信息索引

分布式索引

按文档划分（Document Partitioning）：将文档分为若干个子集，每个子集建立索引

按单词划分（Term partitioning）：将单词字典分为若干个子集，每个子集包含部分单词的全部。一次只能处理一个单词。

由于不同单词出现的频率不同，一般采用文档划分，原因是在扩展性、负载均衡、容错、查询处理方式等方面文档划分都比单词划分更有优势。

如何有效地压缩

倒排索引：由单词词典和单词对应的倒排列表组成。因此，压缩包括对单词词典和倒排列表的压缩。

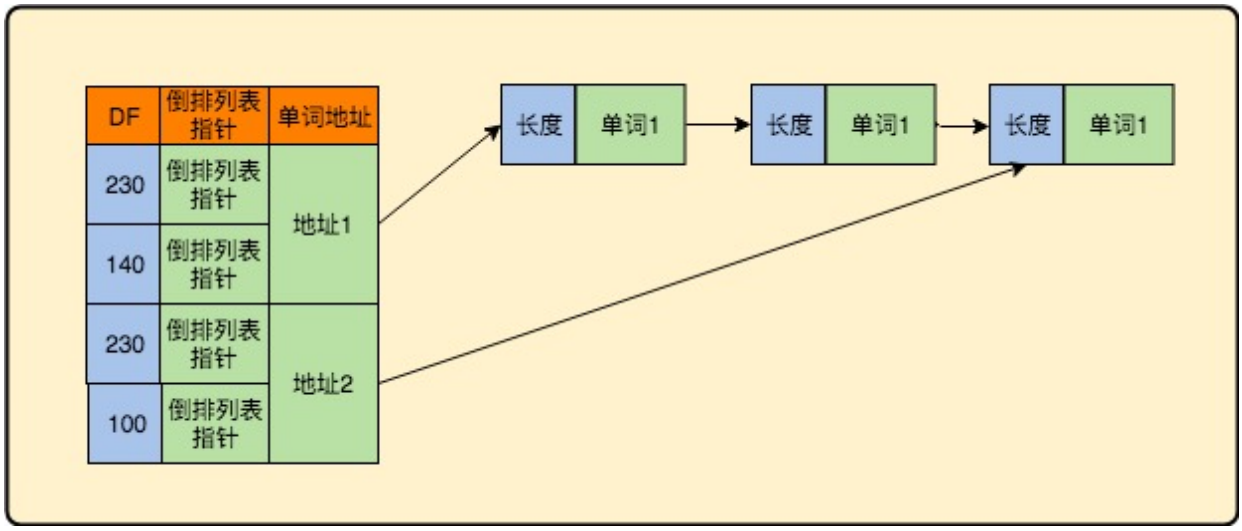
词典的压缩

单词词典的每个单词包含：

1. 单词本身：不定长
2. 文档频率信息（DF）：4 字节

3. 指向倒排列表的指针信息：4 字节

为了解决单词本身长度不一样，可以保存单词内容的指针，这样长度就一样了。而为了避免给每个单词都指明指针，可以将几个单词为一组，用一个指针。各个单词之前加上单词长度。如下图。



实验表明，可以节省 60% 的空间。

倒排列表的压缩

倒排列表由

1. 文档编号：排序，并记录差值，将大数字变为小数字
2. 词频信息：一般为小数字
3. 单词位置信息：本身已经排序，因此只要记录差值，即可实现大数字变为小数字

特点：数字分布不均衡，小数字占了绝大多数

评价指标：按照重要性递增的顺序依次为压缩速度、压缩率、解压速度。

压缩算法基础：

一元编码：数字 5 表示为 111110，数字 3 表示为 1110。仅适用于小数字。

二进制编码：以二进制表示数字

基础以一元编码+二进制编码或二进制编码

无损压缩算法

1. Elias Gamma 算法与 Elias Delta 算法
2. 哈夫曼编码

3. Golomb 算法与 Rice 算法
4. 插值编码
5. 变长字节算法（Variable Byte）
6. Simple Family
7. SimpleX 算法
8. PForDelta 算法

有损压缩算法

1. 以单词为中心的索引剪裁：当文档多于 K 个时，计算单词与其倒排索引文档的相似度分数并排序，所有分数减去最小分数，保留相似度最高的前 K 个，记录第 k 个的相似度，用该相似度乘以 a 作为阈值，删除倒排索引中低于阈值的文档
2. 以文档为中心的索引剪裁：计算文档中单词的重要性，将重要性低的文档删除。

文档编号重排序

因为文档以 D-Gap 的形式保存，因此，为了更好地压缩，每个单词的倒排序列表中文档编号应该更加相近，而为了让相近的文档编号相近，直观的方法就是对文档进行类聚。而文档类聚的一般方法就是对文档的 URL 相近的文档更可能属于同一类。

文档查询

概述

两个矛盾：

1. 用户真实需求与发出查询词之间的矛盾
2. 查询词期望的结果与实际检索结果之间的矛盾

对于矛盾一，从系统角度来说很难解决，因此系统假设用户的输入准确地表达了期望的结果，因此，矛盾主要是2。

对于矛盾 2 有两种方法

1. 查询模型：网页与查询的相关性
2. 链接分析：通过链接分析计算网页的重要性

文档划分4 象限

第一象限：相关文档包含查询词

第二象限：相关文档不包含查询词

第三象限：不相关文档包含查询词

第四象限：不相关文档不包含查询词

目标：尽量提高第一、二象限，抑制第三、四象限

检索质量评估方法

在实现有效的查询之前，研究人员必须有一套切实可行的评估检索质量的方法论和系统才能让检索模型可以更加快速地朝正确的方向发展。

1. 文档与用户搜索请求是否相关
2. 文档是否出现在本次搜索结果中

评估指标

1. 精准率与召回率
2. P@10 指标
3. MAP（Mean Average Precision）

网页排序算法

布尔模型

AND OR 等，缺点非常明显。对于相关文档不包含查询词无能为力。

向量空间模型

以文档的单词为维度，计算查询与文档的相似性

权重：

$IDF_k = log\frac{N}{n_k}$ （区分不同文档的能力）

$W_{Tf} = a + (1-a)log\frac{Tf}{Max(Tf)}$ （文档中单词的频率）

$Weight_{word} = IDF \times Tf$

$cosine(Q_i, D_j) = \frac{\sum_{j=1}^{t} w_{ij} \times q_j}{\sqrt{w_{ij}^2 \times \sum_{j=1}^{t} q_j^2}}$

cosine 的缺点是对长文档抑制太多，缺失理论支撑。

概率检索模型

直接对用户需求建模，思路：相关的概率大于不相干的概率，变为一个二分类问题。

$P(R|D) = \frac{P(D|R)P(R)}{P(D)}$ 文档 D 与查询 R 相关的概率

$P(NR|D) = \frac{P(D|NR) P(NR)}{P(D)}$ 文档 D 与查询 R 不相关的概率

$P(R|D) > P(NR|D)$

$\frac {P(D|R)} {P(D|NR)} > \frac {P(NR)}{P(R)}$

BM25，BM25F 是最好的概率检索模型

基于统计的语言模型

每个文档建立不同的语言模型，结果优于向量模型，与 BM25 等概率模型相当

机器学习

假设用户点击的网页和用户查询非常相关。

当排序依赖的因素很多的时候，人为设计计算公式已经不可能。必须借助机器学习的方式。

“点击训练”：监督学习

1. 人工进行数据标注
2. 人工进行文档特征提取
3. 学习分类函数
4. 实际搜索中应用

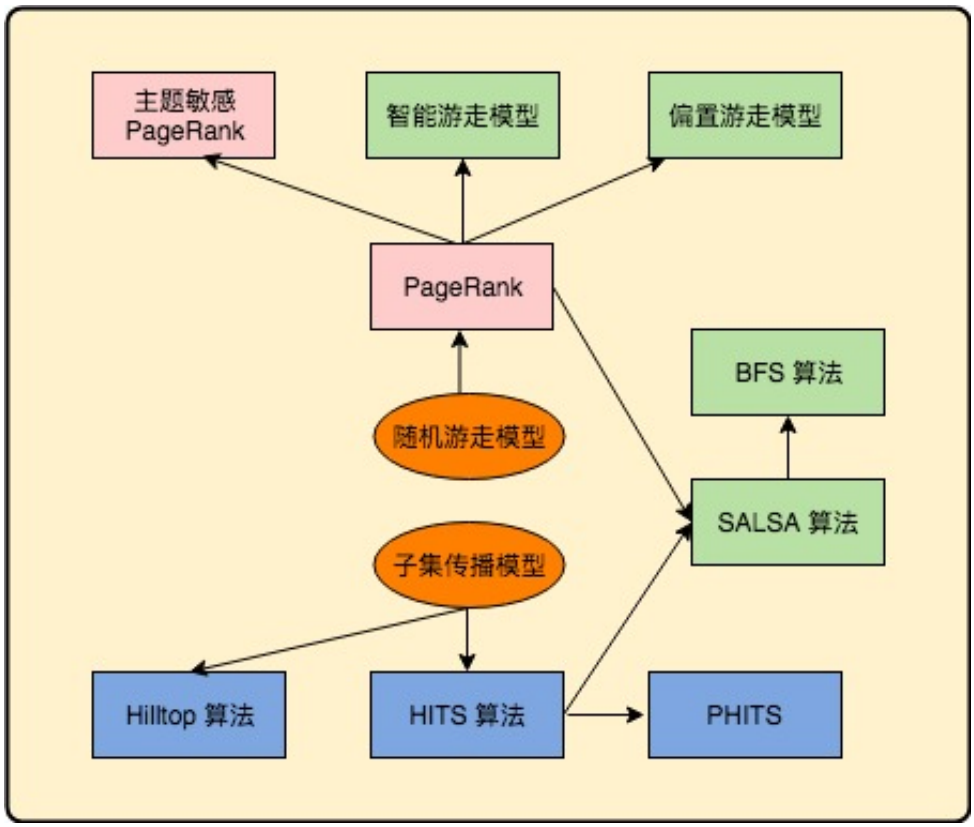
目前的深度学习，不需要人工进行文档特征提取。

1. 单文档方法
2. 文档对方法
3. 文档列表方法

常用链接分析模型

随机游走模型：对直接跳转和远程跳转两种用户浏览行为进行抽象的概念模型

子集传播模型：如何定义子集（有哪些性质）；如何给子集内网页一定的初始分值；如何将分值传播到其他网页。



PageRank

在 PageRank 之前已经有研究者提出入链数量来进行链路分析，而且效果明显，PageRank 同时考虑了 入链数量与网页质量两个因素，获得更好的结果。

1. 数量假设：如果一个网页收到其他网页的入链数量越多，则页面越重要
2. 质量假设：质量越高的页面链向其他页面传递更大的权重。

总结起来可以通俗理解为：一个人的价值除了他有多少朋友之外，还与他朋友的质量有关。

PageRank 刚开始赋予每个网页相同的重要性得分，通过迭代递归计算来更新每个页面节点的 PageRank 得分，直到得分稳定为止。

总结

建立索引

1. 爬虫
2. 建立倒排列表
3. 对单词词典进行压缩
4. 对倒排列表进行无损压缩，首先将倒排索引进行类聚，让相邻的文档编号相近，文档编号以 D-Gap（文档差值编号）的形式保存
5. 以文档为中心进行索引剪裁（有损压缩）
6. 以单词为中心进行索引剪裁（有损压缩）

查询

1. 词法分析，语法分析
2. 计算查询与文档的相关度计算
3. 对结果进行排序
4. 返回结果

Lucene

Index: 建立索引

analysis：词法分析与语言处理形成 Term

queryParse：语法分析

search ：搜索

similarity：打分

store：索引读写

参考

https://www.cnblogs.com/always-online/category/718305.html

https://www.cnblogs.com/forfuture1978/p/3940965.html

https://www.cnblogs.com/forfuture1978/archive/2009/12/14/1623594.html

https://www.cnblogs.com/forfuture1978/p/3944583.html

https://www.cnblogs.com/forfuture1978/archive/2010/06/13/1757479.html

附录

1. 找到索引的文档

比如

文件一：Students should be allowed to go out with their friends, but not allowed to drink beer.

文件二：My friend Jerry went to school to see his students but found them drunk which is not allowed.

2. 文档传给分词组件(Tokenizer)

分词组件(Tokenizer)会做以下几件事情(此过程称为Tokenize)：

1. 将文档分成一个一个单独的单词。
2. 去除标点符号
3. 去除停词(Stop word)

经过分词(Tokenizer) 后得到的结果称为词元(Token)。

在我们的例子中，便得到以下词元(Token)：

“Students”，“allowed”，“go”，“their”，“friends”，“allowed”，“drink”，“beer”，“My”，“friend”，“Jerry”，“went”，“school”，“see”，“his”，“students”，“found”，“them”，“drunk”，“allowed”

> 所谓停词(Stop word)就是一种语言中最普通的一些单词，由于没有特别的意义，因而大多数情况下不能成为搜索的关键词，因而创建索引时，这种词会被去掉而减少索引的大小。对于每一种语言的分词组件(Tokenizer)，都有一个停词(stop word)集合。比如，英语中停词(Stop word)如：“the”,“a”，“this”等。

3. 将得到的词元(Token)传给语言处理组件(Linguistic Processor)

语言处理组件 (linguistic processor) 主要是对得到的词元 (Token) 做一些同语言相关的处理。

对于英语，语言处理组件(Linguistic Processor)一般做以下几点：

1. 变为小写(Lowercase)。
2. 将单词缩减为词根形式，如“cars”到“car”等。这种操作称为：stemming。
3. 将单词转变为词根形式，如“drove”到“drive”等。这种操作称为：lemmatization。

语言处理组件(linguistic processor)的结果称为词(Term)。

在我们的例子中，经过语言处理，得到的词(Term)如下：

“student”，“allow”，“go”，“their”，“friend”，“allow”，“drink”，“beer”，“my”，“friend”，“jerry”，“go”，“school”，“see”，“his”，“student”，“find”，“them”，“drink”，“allow”。

也正是因为有语言处理的步骤，才能使搜索drove，而drive也能被搜索出来。

Stemming 和 lemmatization的异同：

相同之处：Stemming 和 lemmatization都要使词汇成为词根形式。

两者的方式不同：
1. Stemming 采用的是“缩减”的方式：“cars”到“car”，“driving”到“drive”。
2. Lemmatization 采用的是“转变”的方式：“drove”到“drove”，“driving”到“drive”。

两者的算法不同：
1. Stemming 主要是采取某种固定的算法来做这种缩减，如去除“s”，去除“ing”加“e”，将“ational”变为“ate”，将“tional”变为“tion”。
2. Lemmatization 主要是采用保存某种字典的方式做这种转变。比如字典中有“driving”到“drive”，“drove”到“drive”，“am, is, are”到“be”的映射，做转变时，只要查字典就可以了。
3. Stemming 和 lemmatization 不是互斥关系，是有交集的，有的词利用这两种方式都能达到相同的转换。

4. 将得到的词(Term)传给索引组件(Indexer)。

索引组件(Indexer) 主要做以下几件事情：

1. 利用得到的词 (Term) 创建一个字典

Term	Document ID
student	1
allow	1
go	1
their	1
friend	1
allow	1
drink	1
beer	1
my	2

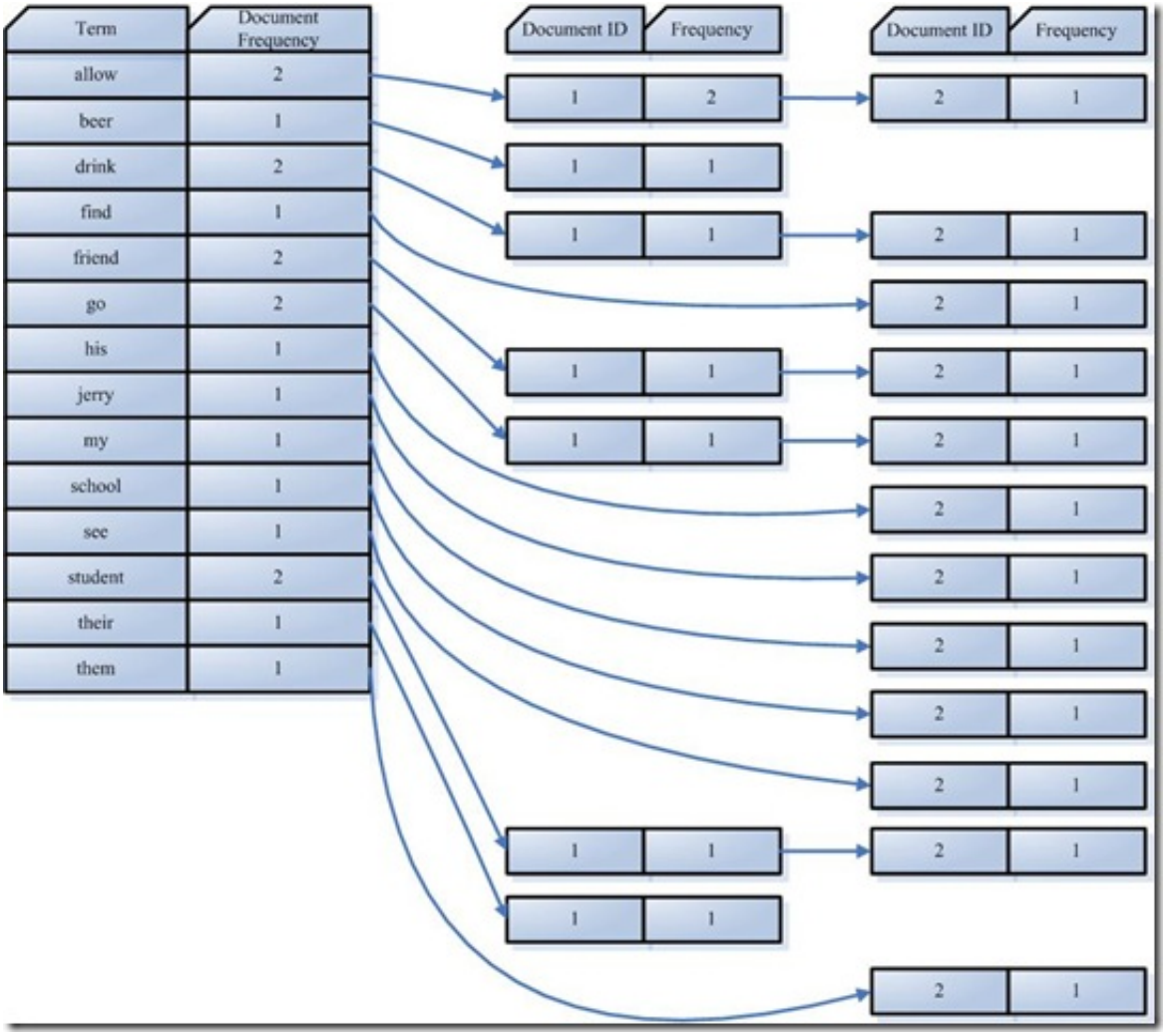
friend	2
jerry	2
go	2
school	2
see	2
his	2
student	2
find	2
them	2
drink	2
allow	2

1. 对字典按字母顺序进行排序。

Term	**Document ID**
allow	1
allow	1
allow	2
beer	1
drink	1
drink	2
find	2
friend	1

friend	2
go	1
go	2
his	2
jerry	2
my	2
school	2
see	2
student	1
student	2
their	1
them	2

2. 合并相同的词(Term)成为文档倒排(Posting List)链表。



在此表中，有几个定义：

- Document Frequency 即文档频次，表示总共有多少文件包含此词(Term)。
- Frequency 即词频率，表示此文件中包含了几个此词(Term)。

所以对词(Term) “allow”来讲，总共有两篇文档包含此词(Term)，从而词(Term)后面的文档链表总共有两项，第一项表示包含“allow”的第一篇文档，即1号文档，此文档中，“allow”出现了2次，第二项表示包含“allow”的第二个文档，是2号文档，此文档中，“allow”出现了1次。

注：真实的搜索引擎，不止有文档编号，还有词频，文档偏移等。

到此为止，索引已经创建好了，我们可以通过它很快的找到我们想要的文档。

而且在此过程中，我们惊喜地发现，搜索“drive”，“driving”，“drove”，“driven”也能够被搜到。因为在我们的索引中，“driving”，“drove”，“driven”都会经过语言处理而变成“drive”，在搜索时，如果您输入“driving”，输入的查询语句同样经过我们这里的一到三步，从而变为查询“drive”，从而可以搜索到想要的文档。