

# 第**5**章

## 非结构化数据处理

**IR** 布尔检索 倒排表 模糊检索 相似性 **TFIDF**

# 信息检索

# INFORMATION RETRIEVAL

**Information Retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).**

信息检索是从大规模非结构化数据（通常是文本）的集合（通常保存在计算机上）中找出满足用户信息需求的资料（通常是文档）的过程。

**Document – 文档**

**Unstructured – 非结构化**

**Information need – 信息需求**

**Collection—文档集、语料库**

# IR VS 数据库

## 非结构化数据 VS 结构化

结构化数据即指“表”中的数据

Employee	Manager	Salary
Smith	Jones	50000
Chang	Smith	60000
Ivy	Smith	50000

数据库常常支持范围或者精确匹配查询。e.g.,  
*Salary < 60000 AND Manager = Smith.*

# 非结构化数据

通常指自由文本

允许

- 关键词加上操作符号的查询
- 更复杂的概念性查询
  - 找出所有的有关药物滥用(drug abuse)的网页

经典的检索模型一般都针对自由文本进行处理

# 半结构化数据

没有数据是完全无结构的

**<title>李甲主页</title>**

**<body>...</body> ...**

半结构化查询

- Title contains data AND Bullets contain search
  - ... 这里还没有提文本的语言结构

# IR中的基本假设

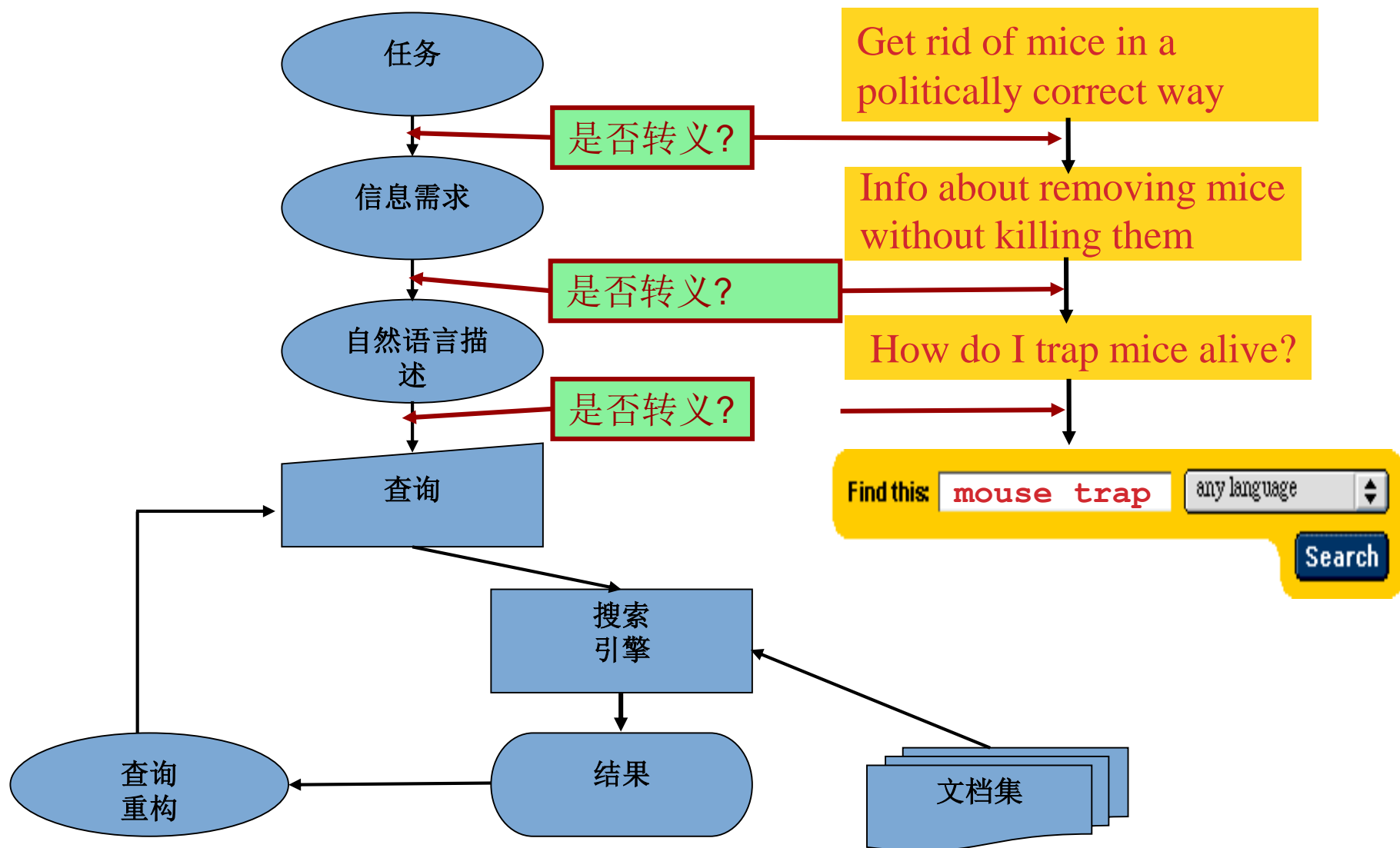
文档集**Collection**: 由固定数目的文档组成

目标: 返回与用户需求相关的文档并辅助用户来完成某项任务

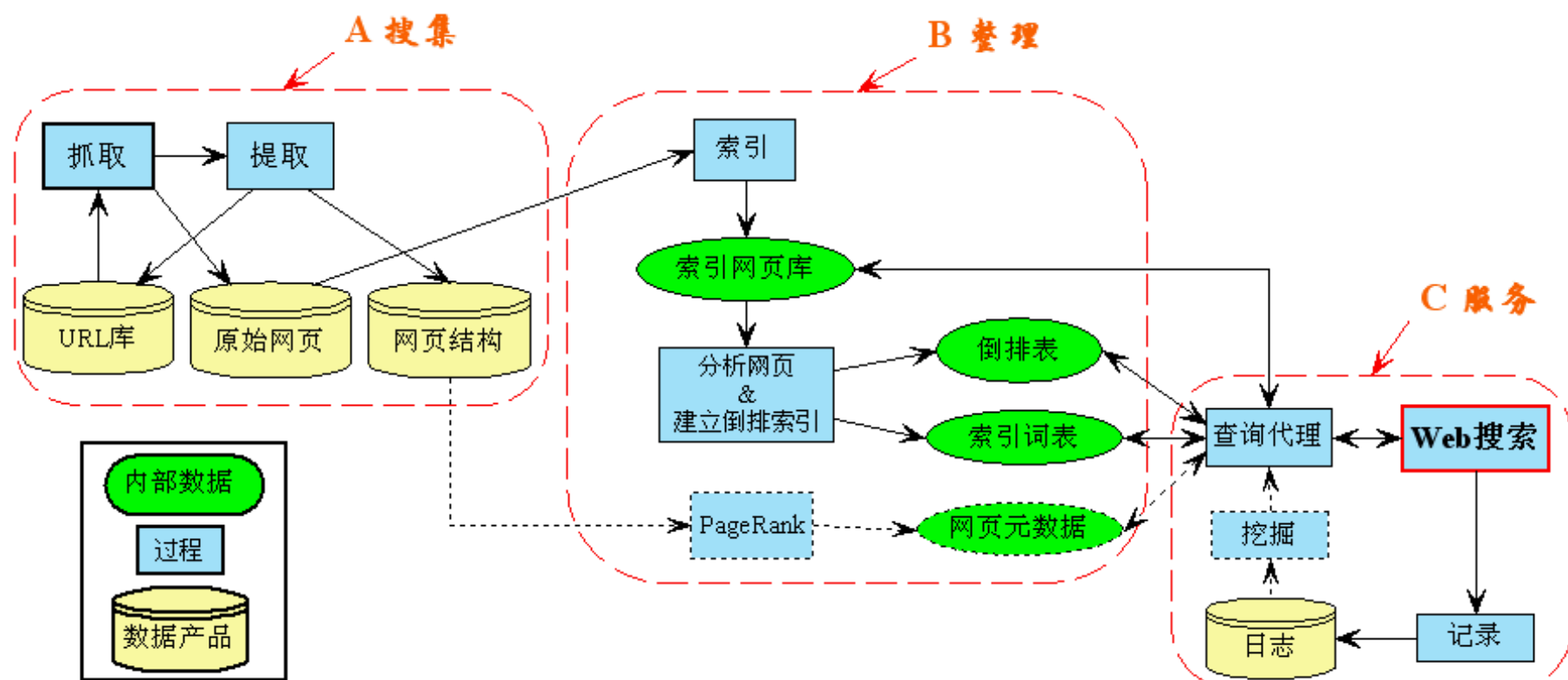
相关性**Relevance**:

- 主观的概念
- 反映对象的匹配程度
- 不同应用相关性不同

# 典型的搜索过程



# 通用搜索引擎系统流程





# 信息检索模型的分类

- 从所使用的数学方法上分：
  - 基于集合论的IR模型(Set Theoretic models)
    - 布尔模型
    - 基于模糊集模型、扩展布尔模型
  - 基于代数论的IR模型(Algebraic models)
    - 向量空间模型
    - LSI（隐性语义检索）模型
    - 神经网络模型
  - 基于概率统计的IR模型(Probabilistic models)
    - 概率模型
  - 回归模型、语言模型建模IR模型、推理网络模型、信任度网络模型

# 检索效果的评价

**正确率(Precision)** : 返回结果文档中正确的比例。如返回**80**篇文档，其中**20**篇相关，正确率**1/4**。

**召回率(Recall)** : 全部相关文档中被返回的比例，如返回**80**篇文档，其中**20**篇相关，但是总的应该相关的文档是**100**篇，召回率**1/5**。

正确率和召回率反映检索效果的两个方面，缺一不可。

- 全部返回，正确率低，召回率100%
- 只返回一个非常可靠的结果，正确率100%，召回率低

# 布尔检索

针对布尔查询的检索，布尔查询是指利用 **AND, OR** 或者 **NOT** 操作符将词项连接起来的查询。

# 一个简单的例子(《莎士比亚全集》)

莎士比亚的哪部剧本包含**Brutus**及**Caesar**但是不包含**Calpurnia**? 布尔表达式为 **Brutus AND Caesar AND NOT Calpurnia**。

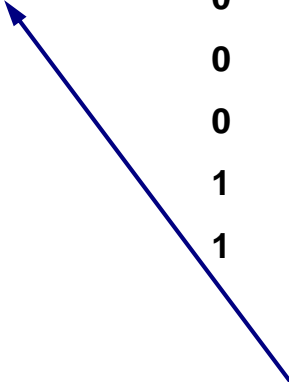
笨方法： 从头到尾扫描所有剧本，对每部剧本判断它是否包含**Brutus AND Caesar**，同时又不包含**Calpurnia**

笨方法为什么不好？

- 速度超慢 (特别是大型文档集)
- 处理**NOT Calpurnia** 并不容易 (一旦包含即可停止)
- 不太容易支持其他操作 (e.g., find the word **Romans** near **countrymen**)
- 不支持检索结果的排序 (即只返回较好的结果)

# 词项-文档(TERM-DOC)的关联矩阵

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0



***Brutus AND Caesar BUT NOT  
Calpurnia***

若某剧本包含某单词，则该位置上为1，否则为0

# 关联向量(INCIDENCE VECTORS)

关联矩阵的每一列都是 0/1 向量，每个0/1都对应一个词项

给定查询Brutus AND Caesar AND NOT Calpurnia

取出三个列向量，并对Calpurnia 的列向量求补，最后按位进行与操作

**110100 AND 110111 AND 101111 = 100100.**

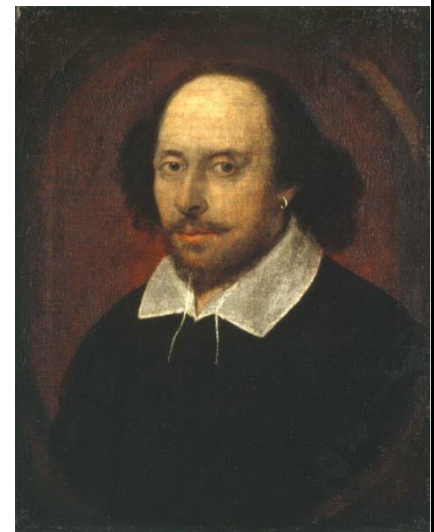
# 上述查询的结果文档

**Antony and Cleopatra, Act III, Scene ii**

**Agrippa [Aside to DOMITIUS ENOBARBUS]: Why, Enobarbus,  
When Antony found Julius Caesar dead,  
He cried almost to roaring; and he wept  
When at Philippi he found Brutus slain.**

**Hamlet, Act III, Scene ii**

**Lord Polonius: I did enact Julius Caesar I was killed  
i' the Capitol; Brutus killed me.**



# 大文档集

假定  $N = 1$  百万篇文档(1M), 每篇有1000个词(1K)

假定每个词平均有6个字节(包括空格和标点符号)

- 那么所有文档将约占6GB 空间.

假定 词汇表的大小(即词项个数)  $M = 500K$

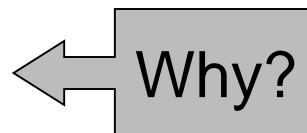


# 词项-文档矩阵将非常大

矩阵大小为 **500K x 1M=500G**

但是该矩阵中最多有**10亿(1G)**个1

- 词项-文档矩阵高度稀疏(sparse).
- 稀疏矩阵



应该有更好的表示方式

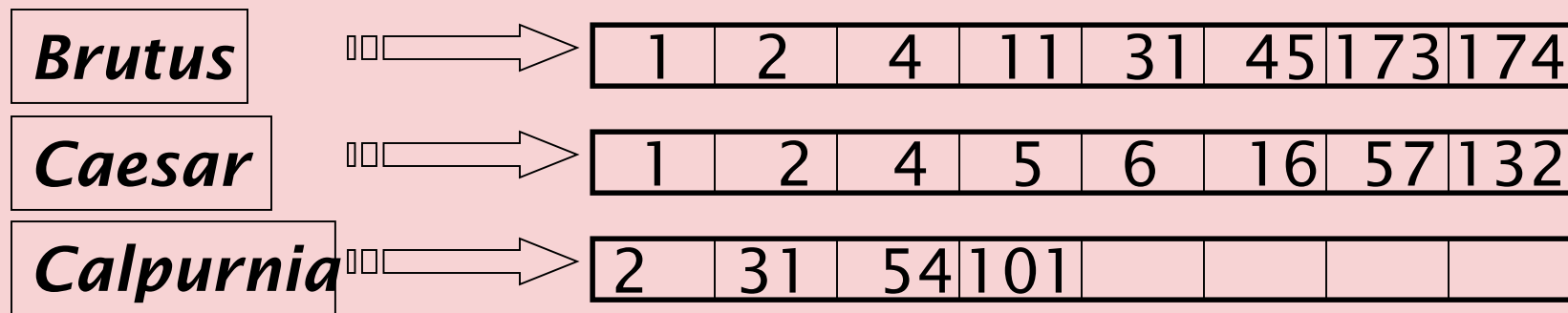
- 比如我们仅仅记录所有1的位置

# 倒排索引(INVERTED INDEX)

对每个词项t, 记录所有包含t的文档列表.

- 每篇文档用一个唯一的 docID来表示, 通常是正整数, 如1,2,3...

能否采用定长数组的方式来存储docID列表



文档14中加入单词**Caesar**时该如何处理?

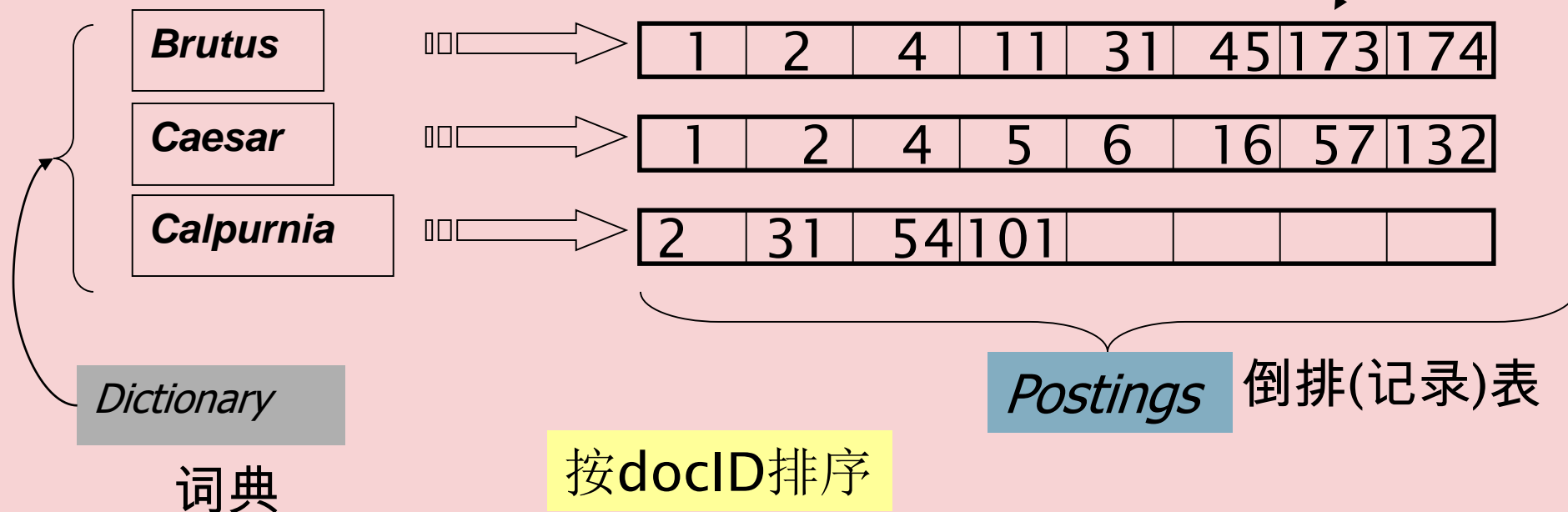
# 倒排索引(续)

## 通常采用变长表方式

- 磁盘上，顺序存储方式比较好，便于快速读取
- 内存中，采用链表或者可变长数组方式
  - 存储空间/易插入之间需要平衡

倒排记录

Posting



# 倒排索引构建

待索引文档



Friends, Romans, countrymen.

⋮

词流

Tokenizer

词条化工具

Friends

Romans

Countrymen

修改后的词条

Linguistic  
modules

语言分析工具

friend

roman

countryman

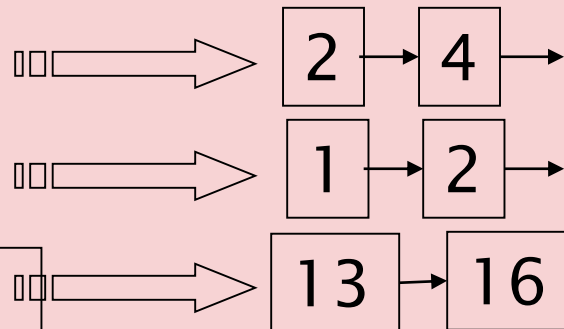
倒排索引

Indexer

*friend*

*roman*

*countryman*



# 文档分析

文档格式处理

- pdf/word/excel/html?

文档语言识别

文档编码识别

文档语言识别和编码识别理论上都可以看成分类问题，基于分类方法可以处理。但是实际中，常常采用启发式方法.....

# 词条化

输入: “**Friends, Romans and Countrymen**”

输出: 词条(Token)

- Friends
- Romans
- Countrymen

词条就是一个字符串实例，词条在经过进一步处理之后将放入倒排索引中的词典中。

词条化中的问题: 词条如何界定?

# 词条化

## 一系列问题:

- Finland's capital →
- Finland? Finlands? Finland's?
- Hewlett-Packard → 看成Hewlett 和 Packard 两个词条?
  - state-of-the-art
  - co-education
  - lowercase, lower-case, lower case ?
- San Francisco: 到底是一个还是两个词条?
  - 如何判断是一个词条?

# 词条归一化成词项

将文档和查询中的词归一化成同一形式：

- U.S.A. 和 USA

归一化的结果就是词项，而词项就是我们最终要索引的对象。

可以采用隐式规则的方法来表示多个词条可以归一成同一词项：

- 剔除句点
  - U.S.A., USA: USA
- 剔除连接符
  - anti-discriminatory, antidiscriminatory: antidiscriminatory



# 大小写问题

可以将所有字母转换成小写形式

- 例外: 句中的大写单词?
  - e.g., General Motors (GM, 通用公司)
  - Fed (美联储)vs. fed(饲养)
  - SAIL (印度钢铁管理局) vs. sail(航行)
- 通常情况下将所有字母转成小写是一种很合适的方式, 因为用户倾向于用小写方式输入

**Google**的例子:

- 查询 C.A.T.
- 排名第一的结果是“cat”而不是 Caterpillar Inc.

# 同义词词典及**SOUNDEX**方法

## 同义词和同音/同形异义词的处理

- E.g., 手动建立词典，记录这些词对
  - car = automobile      color = colour
- 利用上述词典进行索引
  - 当文档包含 automobile时，利用car-automobile进行索引
- 或者对查询进行扩展
  - 当查询包含 automobile时，同时也查car

## 拼写错误的处理(**Clinton**→**Klinten**)

- 一种解决方法是soundex方法，基于发音建立词之间的关系

# 词形归并

将单词的屈折变体形式还原为原形

例子：

- am, are, is → be
- car, cars, car's, cars' → car
- the boy's cars are different colors → the boy car be different color

词性归并意味中将单词的变形形式“适当”还原成一般词典中的单词形式

- found → find? found?

# 词干还原

将词项归约(**reduce**)成其词干(**stem**), 然后再索引

“词干还原”意味着词缀的截除

- 与语言相关
- 比如, 将 automate(s), automatic, automation都还原成 automat

***for example compressed and compression are both accepted as equivalent to compress.***



for exampl compress and  
compress are both accept  
as equival to compress

# PORTER算法

英语词干还原中最常用的算法

规定: 选择可应用规则组中包含最长词缀的规则

- SSES→SS                      caresses→caress
- S→                              cats→cat
- ies → I
- ational → ate
- tional → tion

规则适用条件的表达

- (m>1) EMENT →
  - replacement → replac
  - cement → cement

# MARTIN PORTER

英国人，剑桥大学

2000年度 Tony Kent Strix award得主

- 信息检索领域一个著名的奖项



**Porter's stemmer**, 有很多语言的版本

**Snowball** 工具，支持多种语言的**stemming**(法语、德语、葡萄牙语、西班牙语挪威语等等)

# 短语查询 **PHRASE QUERIES**

输入查询作为一个短语整体，比如 “**stanford university**” “中国科学院”。

因此，句子 “**I went to university at Stanford**” 就不应该是答案 （“我去了中国 农业 科学院”）

- 有证据表明，用户很容易理解短语查询的概念，这也是很多搜索引擎“高级搜索”中比较成功的一个功能。
- 但是很多查询是隐式短语查询， **information retrieval textbook** → **[information retrieval] textbook**

这种情况下，倒排索引仅仅采用如下方式是不够的

**term + docIDs**

# 第一种做法: 双词(**BIWORD**)索引

每两个连续的词组成词对(作为短语)来索引

比如文本片段 “**Friends, Romans, Countrymen**” 会产生两个词对

- friends romans
- romans countrymen

索引构建时，将每个词对看成一个词项放到词典中

这样的话，两个词组成的短语查询就能直接处理



# 更长的短语查询处理

例子： **stanford university palo alto**， 处理方法： 将其拆分成基于双词的布尔查询式：

**stanford university AND university palo AND palo alto**

如果不检查文档，无法确认满足上述表达式的文档是否真正满足上述短语查询，也就是说满足上述布尔表达式只是满足短语查询的必要条件。



很难避免伪正例的出现！

# 扩展的双词 (**EXTENDED BIWORD**)

对待索引文档进行词性标注

将词项进行组块，每个组块包含名词 (**N**) 和冠词/介词 (**X**)

称具有**NX\*N**形式的词项序列为扩展双词(**extended biword**)

- 将这样扩展词对作为词项放入词典中

例子: **catcher in the rye** (书名: 麦田守望者)

- N            X   X   N

查询处理: 将查询也分析成 **N**和**X**序列

- 将查询切分成扩展双词
- 在索引中查找: **catcher rye**

# 关于双词索引

1. 会出现伪正例子
2. 由于词典中词项数目剧增，导致索引空间也激增
  - 如果3词索引，那么更是空间巨大，无法忍受
3. 双词索引方法并不是一个标准的做法 (即倒排索引中一般不会全部采用双词索引方法)，但是可以和其他方法混合使用

## 第二种解决方法: 带位置信息索引 (POSITIONAL INDEXES)

在倒排记录表中, 对每个term在每篇文档中的每个位置(偏移或者单词序号)进行存储:

- <term, 出现term的文档篇数;
- doc1: 位置1, 位置2 ... ;
- doc2: 位置1, 位置2 ... ;
- 等等>

# 位置索引的例子

对于输入的短语查询，需要在文档的层次上进行迭代  
(不同位置上)合并，不仅仅简单合并，还要考虑位置匹  
配

<*be*: 993427;

*1*: 7, 18, 33, 72, 86, 231;

*2*: 3, 149;

*4*: 17, 191, 291, 430, 434;

*5*: 363, 367, ...>



1,2,4,5这几篇文章  
中哪篇包含 “*to be  
or not to be*”?

# 短语查询的处理

短语查询：“to be or not to be”

对每个词项，抽出其对应的倒排记录表: to, be, or, not.

合并<docID:位置>表，考虑“to be or not to be”.

- to:
  - 2:1,17,74,222,551; 4:8,16,190,429,433; 7:13,23,191; ...
- be:
  - 1:17,19; 4:17,191,291,430,434; 5:14,19,101; ...

邻近搜索中的搜索策略与此类似，不同的是此时考虑前后位置之间的距离不大于某个值

# 邻近式查询(**PROXIMITY QUERY**)

**LIMIT /3 STATUTE /3 FEDERAL /2 TORT**

- /k 表示 “在 k 个词之内”

很明显，位置索引可以处理邻近式查询，而双词索引却不能

# 位置索引的大小

1. 位置索引增加了位置信息，因此空间较大，但是可以采用索引压缩技术进行处理
2. 当然，相对于没有位置信息的索引，位置索引的存储空间明显大于无位置信息的索引
3. 另外，位置索引目前是实际检索系统的标配，这是因为实际中需要处理短语(显式和隐式)和邻近式查询



# 位置索引的大小

词项在每篇文档中的每次出现都需要一个存储单元

因此索引的大小依赖于文档的平均长度

- 平均Web页面的长度 <1000 个词项
- 美国证监会文件(SEC filings), 书籍, 甚至一些史诗 ... 和容易就超过 100,000 个词项

假定某个词项的出现频率是**0.1%**

文档大小	倒排记录表的数目	位置索引存储单元
1000	1	1
100,000	1	100

# 一些经验规律

位置索引的大小大概是无位置信息索引的**2-4**倍

位置索引大概是原始文本容量的**35-50%**

提醒：上述经验规律适用于英语及类英语的语言

# 混合索引

上述两种索引方式可以混合使用

- 对某些特定的短语 (如“Michael Jackson”, “Britney Spears”), 如果采用位置索引的方式那么效率不高
  - 还有“The Who” (英国一著名摇滚乐队), 采用位置索引, 效率更低

**Williams et al. (2004)**对一种混合的索引机制进行了评估

- 采用混合机制, 那么对于典型的Web查询(比例)来说, 相对于只使用位置索引而言, 仅需要其 $\frac{1}{4}$  的时间
- 相对于只使用位置索引, 空间开销只增加了26%

# 索引构建过程: 词条序列

<词条, docID>二元组

Doc 1

I did enact Julius  
Caesar I was killed  
i' the Capitol;  
Brutus killed me.

Doc 2

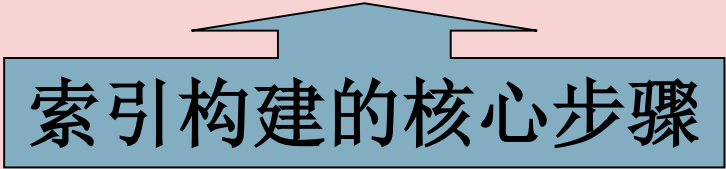
So let it be with  
Caesar. The noble  
Brutus hath told you  
Caesar was ambitious



Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

# 索引构建过程: 排序

按词项排序，相同词项按docID排序



Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2



Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

# 索引构建过程: 词典 & 倒排记录表

某个词项在单篇文章中的多次出现会被合并

拆分成词典和倒排记录表两部分

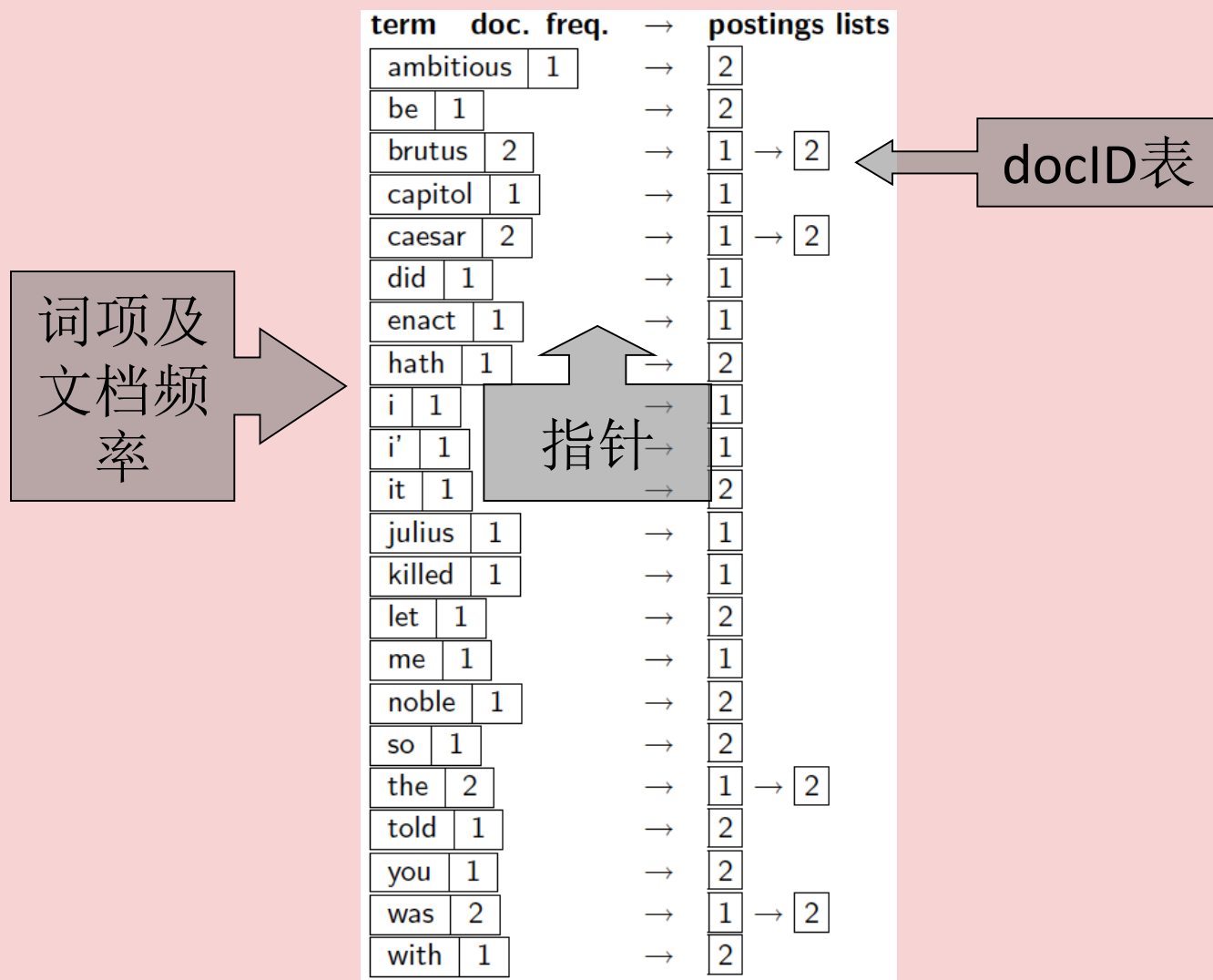
每个词项出现的文档数目(**doc. frequency, DF**)会被加入

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2



term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

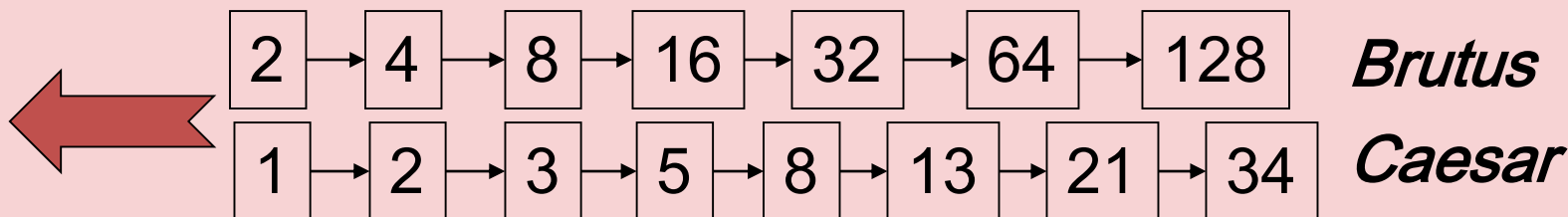
# 存储开销计算



# AND查询的处理

考虑如下查询（从简单的布尔表达式入手）：

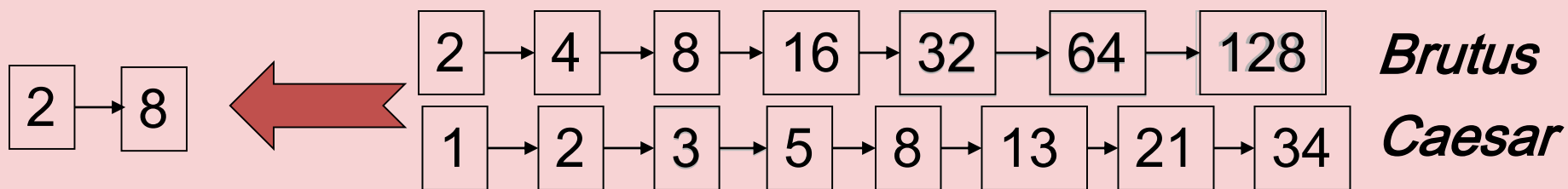
- Brutus AND Caesar
- 在词典中定位 Brutus
  - 返回对应倒排记录表(对应的docID)
- 在词典中定位 Caesar
  - 再返回对应倒排记录表
- 合并(Merge)两个倒排记录表，即求交集





# 合并过程

每个倒排记录表都有一个定位指针，两个指针同时从前往后扫描，每次比较当前指针对应倒排记录，然后移动某个或两个指针。合并时间为两个表长之和的线性时间



假定表长分别为 $x$  和  $y$ , 那么上述合并算法的复杂度为  $O(x+y)$

关键原因: 倒排记录表按照docID排序

# 其它布尔查询的处理

- OR表达式: Brutus AND Caesar
- 两个倒排记录表的并集
- NOT表达式: Brutus AND NOT Caesar
- 两个倒排记录表的减

一般的布尔表达式

**(Brutus OR Caesar) AND NOT**

**(Antony OR Cleopatra)**

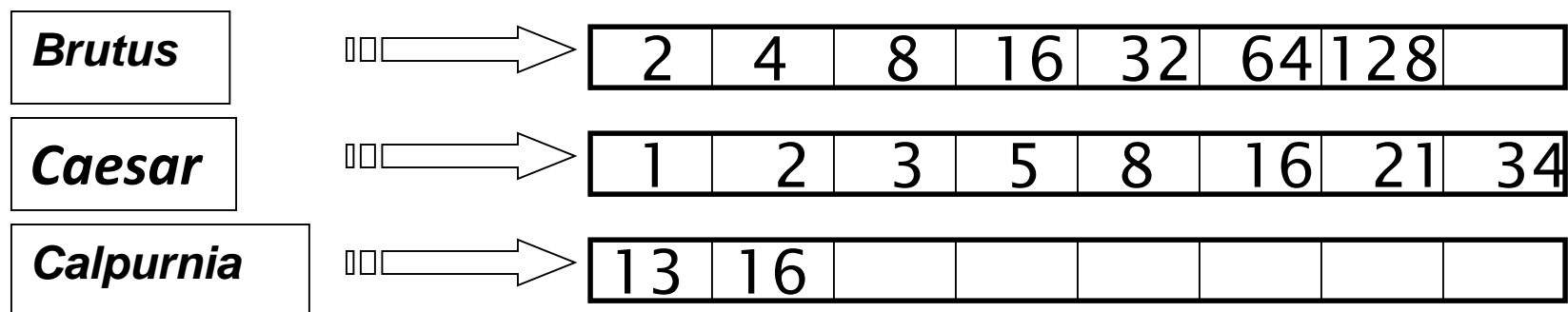
查询处理的效率问题！

# 查询优化

查询处理中是否存在处理的顺序问题？

考虑  $n$  个词项的 **AND**

对每个词项，取出其倒排记录表，然后两两合并



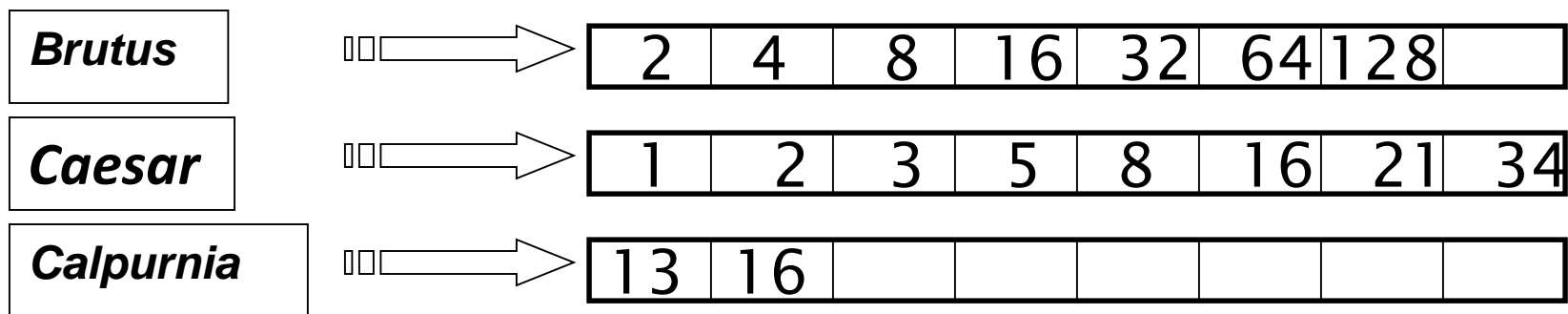
**查询:** *Brutus AND Calpurnia AND Caesar*

# 查询优化

按照表从小到大(即df从小到大)的顺序进行处理:

- 每次从最小的开始合并

这是为什么保存  
df的原因之一



相当于处理查询 (***Calpurnia AND Brutus***) ***AND Caesar***.

# 更通用的优化策略

**e.g., (madding OR crowd) AND (ignoble OR strife)**

- 每个布尔表达式都能转换成上述形式(合取范式)

获得每个词项的**df**

(保守)通过将词项的**df**相加，估计每个**OR**表达式对应的倒排记录表的大小

按照上述估计从小到大依次处理每个**OR**表达式.

# 布尔检索的优点

构建简单，或许是构建IR系统的一种最简单方式

- 在30多年中是最主要的检索工具
- 当前许多搜索系统仍然使用布尔检索模型：
  - 电子邮件、文献编目、Mac OS X Spotlight工具

# 布尔检索例子: **WESTLAW**

(付费用户数目)最大的商业化法律搜索服务引擎 (1975年开始提供服务; 1992年加入排序功能)

几十T数据, **700,000**用户

大部分用户仍然使用布尔查询

查询的例子:

- 有关对政府侵权行为进行索赔的诉讼时效(What is the statute of limitations in cases involving the federal tort claims act?)
- **LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM**
  - /3 = within 3 words, /S = in same sentence

# 布尔检索例子: **WESTLAW**

另一个例子:

- 残疾人士能够进入工作场所的要求 (Requirements for disabled people to be able to access a workplace)
- disabl! /p access! /s work-site work-place (employment /3 place

扩展的布尔操作符

很多专业人士喜欢使用布尔搜索

- 非常清楚想要查什么、能得到什么

但是这并不意味着布尔搜索其实际效果就很好....



# GOOGLE支持布尔查询

想查关于**2013年快男 6进5** 比赛的新闻，用布尔表达式怎么构造查询？

**(2013 OR 去年) AND (快乐男声 OR 快男OR 快乐男生)  
AND (6进5 OR 六进五 OR 六 AND 进 AND 五)**

表达式相当复杂，构造困难！

不严格的话结果过多，而且很多不相关；非常严格的话结果会很少，漏掉很多结果。

# 布尔检索的缺点

布尔查询构建复杂，不适合普通用户。构建不当，检索结果过多或者过少

没有充分利用词项的频率信息

- 1 vs. 0 次出现
- 2 vs. 1次出现
- 3 vs. 2次出现, ...
- 通常出现的越多越好，需要利用词项在文档中的词项频率(term frequency, tf)信息

不能对检索结果进行排序

# 排序式检索

# 排序式检索(Ranked retrieval)

- 迄今为止，我们主要关注的是布尔查询
  - 文档要么匹配要么不匹配
- 对自身需求和文档集性质非常了解的专家，布尔查询是不错的选择
- 对应用开发来说也非常简单，很容易就可以返回**1000**多条结果
- 然而对大多数用户来说不方便
- 大部分用户不能撰写布尔查询或者他们认为需要大量训练才能撰写合适的布尔查询
- 大部分用户不愿意逐条浏览**1000**多条结果，特别是对**Web**搜索更是如此

# 布尔搜索的不足：结果过少或者过多

- 布尔查询常常会导致过少( $=0$ )或者过多( $>1000$ )的结果
- 在布尔检索中，需要大量技巧来生成一个可以获得合适规模结果的查询

# 排序式检索

- 排序式检索可以避免产生过多或者过少的结果
- 大规模的返回结果可以通过排序技术来避免
- 只需要显示前**n**条结果，比如**10**条
- 不会让用户感觉到信息太多
- 前提：排序算法真的有效，即相关度大的文档结果会排在相关度小的文档结果之前

# 排序式检索中的评分技术

- 我们希望，在同一查询下，文档集中相关度高的文档排名高于相关度低的文档
- 如何实现？
- 通常做法是对每个查询-文档对赋一个 $[0, 1]$ 之间的分值
- 该分值度量了文档和查询的匹配程度

# 词袋(**Bag of words**)模型

- 不考虑词在文档中出现的顺序
- ***John is quicker than Mary* 及 *Mary is quicker than John*** 的表示结果一样
- 这称为一个词袋模型(**bag of words model**)
- 在某种意思上说，这种表示方法是一种“倒退”，因为位置索引中能够区分上述两篇文档



# 第一种方法: Jaccard系数

- 计算两个集合重合度的常用方法
- 令 **A** 和 **B** 为两个集合
- **Jaccard**系数的计算方法:

$$\text{JACCARD}(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad (A \neq \emptyset \text{ or } B \neq \emptyset)$$

- **JACCARD (A, A) = 1**
- **JACCARD (A, B) = 0** 如果 **A**  $\cap$  **B** = 0
- **A** 和 **B** 不一定要同样大小
- **Jaccard** 系数会给出一个**0**到**1**之间的值

# Jaccard系数的计算样例

- 查询 “**ides of March**”
- 文档 “**Caesar died in March**”

$$\text{JACCARD}(q, d) = 1/6$$

# 课堂练习

- 计算下列查询-文档之间的**Jaccard**系数
  1. q: [information on cars] d: “all you’ve ever wanted to know about cars”
  2. q: [information on cars] d: “information on trucks, information on planes, information on trains”
  3. q: [red cars and red trucks] d: “cops stop red cars more often”

# Jaccard系数的不足

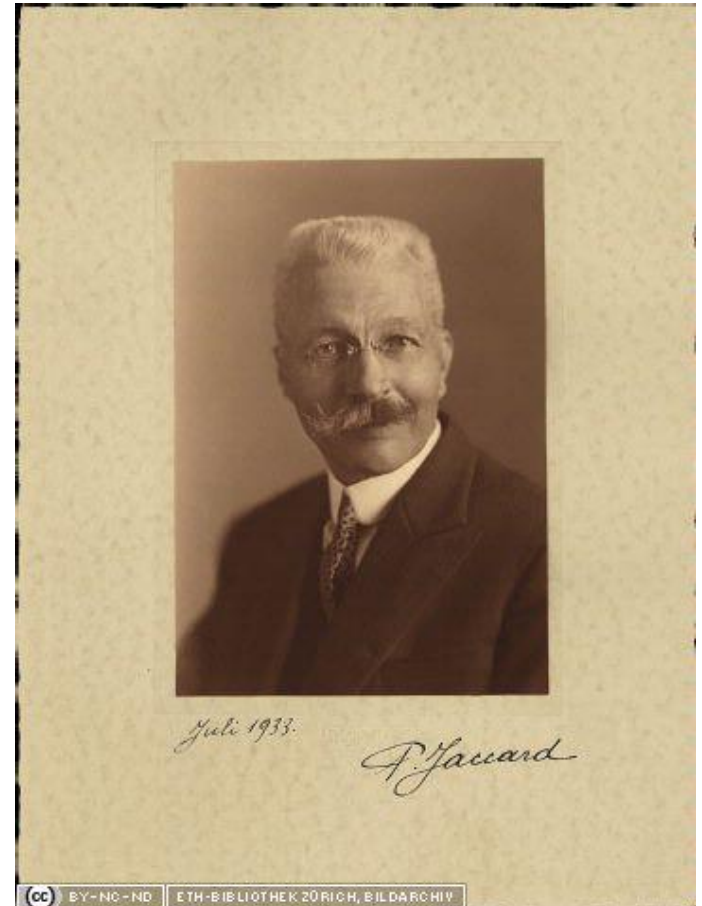
- 不考虑词项频率，即词项在文档中的出现次数
- 罕见词比高频词的信息量更大，**Jaccard**系数没有考虑这个信息
- 没有仔细考虑文档的长度因素
- 本讲义后面，我们将使用  $|A \cap B| / \sqrt{|A \cup B|}$  (即余弦计算) 来代替  $|A \cap B| / |A \cup B|$ ，前者进行的长度归一化

# PAUL JACCARD(1868-1944)

瑞士植物学家，ETH教授

1894年毕业于苏黎世联邦理工学院ETH(出过包括爱因斯坦在内的21位诺贝尔奖得主)

1901年提出Jaccard Index即Jaccard Coefficient概念



# 查询-文档匹配评分计算

- 如何计算查询-文档的匹配得分？最直观的方法：
- 先从单词项查询开始
- 若该词项不出现在文档当中，该文档得分应该为**0**
- 该词项在文档中出现越多，则得分越高

# 词项频率 **tf**

- 词项  $t$  的词项频率  $\mathbf{tf}_{t,d}$  是指  $t$  在  $d$  中出现的次数
- 下面将介绍利用 **tf** 来计算文档评分的方法
- 第一种方法是采用原始的 **tf** 值 (**raw tf**)
- 但是原始 **tf** 不太合适：
  - 某个词项在 **A** 文档中出现十次，即  $\mathbf{tf} = 10$ ，在 **B** 文档中  $\mathbf{tf} = 1$ ，那么 **A** 比 **B** 更相关
  - 但是相关度不会相差 **10** 倍
- 相关度不会正比于词项频率 **tf**

# 一种替代原始**tf**的方法: **对数词频**

- **t** 在 **d** 中的对数词频权重定义如下:

$$w_{t,d} = \begin{cases} 1 + \log_{10} \text{tf}_{t,d} & \text{if } \text{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$$

- **tf**<sub>*t,d*</sub> → **w**<sub>*t,d*</sub> :

**0** → **0**, **1** → **1**, **2** → **1.3**, **10** → **2**, **1000** → **4**, 等等

- **文档-词项的匹配得分**是所有同时出现在 **q** 和文档 **d** 中的词项的对数词频之和

- $\sum_{t \in q \cap d} (1 + \log \text{tf}_{t,d})$

- 如果两者没有公共词项, 则得分为**0**



# 文档中的词频 **vs.** 文档集中的词频

- 除词项频率**tf**之外，我们还想利用词项在整个文档集中的频率进行权重和评分计算

# 罕见词项所期望的权重

- 罕见词项比常见词所蕴含的信息更多
- 考虑查询中某个词项，它在整个文档集中非常罕见 (例如 **ARACHNOCENTRIC**).
- 某篇包含该词项的文档很可能相关
- 于是，我们希望像**ARACHNOCENTRIC**一样的罕见词项将有较高权重

# 常见词项所期望的权重

- 常见词项的信息量不如罕见词
- 考虑一个查询词项，它频繁出现在文档集中 (如 **GOOD, INCREASE, LINE** 等等)
- 一篇包含该词项的文档当然比不包含该词项的文档的相关度要高
- 但是，这些词对于相关度而言并不是非常强的指示词
- 于是，对于诸如 **GOOD**、**INCREASE** 和 **LINE** 的频繁词，会给一个正的权重，但是这个权重小于罕见词权重

# 文档频率(**Document frequency, df**)

- 对于罕见词项我们希望赋予高权重
- 对于常见词我们希望赋予正的低权重
- 接下来我们使用文档频率**df**这个因子来计算查询-文档的匹配得分
- 文档频率指出现词项的文档数目

# idf 权重

- $df_t$  是出现词项  $t$  的文档数目
- $df_t$  是和词项  $t$  的信息量成反比的一个值
- 于是可以定义词项  $t$  的 **idf** 权重:

$$idf_t = \log_{10} \frac{N}{df_t}$$

(其中  $N$  是文档集中文档的数目)

- $idf_t$  是反映词项  $t$  的信息量的一个指标
- 实际中往往计算  $[\log N / df_t]$  而不是  $[N / df_t]$ ，这可以对 **idf** 的影响有所抑制
- 值得注意的是，对于 **tf** 和 **idf** 我们都采用了对数计算方式

# idf的计算样例

- 利用右式计算 $idf_t$ :

$$idf_t = \log_{10} \frac{1,000,000}{df_t}$$

词项	$df_t$	$idf_t$
calpurnia	1	6
animal	100	4
sunday	1000	3
fly	10,000	2
under	100,000	1
the	1,000,000	0

# idf对排序的影响

- **idf** 会影响至少包含**2**个词项的查询的文档排序结果
- 例如，在查询 “**arachnocentric line**”中，**idf**权重计算方法会增加**ARACHNOCENTRIC**的相对权重，同时降低 **LINE**的相对权重
- 对于单词项查询，**idf**对文档排序基本没有任何影响

# tf-idf权重计算

- 词项的**tf-idf**权重是**tf**权重和**idf**权重的乘积

$$w_{t,d} = (1 + \log \text{tf}_{t,d}) \cdot \log \frac{N}{\text{df}_t}$$

- 信息检索中最出名的权重计算方法
- 注意：上面的 “**·**”是连接符，不是减号
- 其他叫法：**tf.idf**、**tf x idf**



# tf-idf小结

- 词项**t**在文档**d**中的权重可以采用下式计算

$$w_{t,d} = (1 + \log \text{tf}_{t,d}) \cdot \log \frac{N}{\text{df}_t}$$

- **tf-idf**权重
  - 随着词项频率的增大而增大
  - 随着词项罕见度的增加而增大

# 二值关联矩阵

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth ...
ANTHONY	1	1	0	0	0	1
BRUTUS	1	1	0	1	0	0
CAESAR	1	1	0	1	1	1
CALPURNIA	0	1	0	0	0	0
CLEOPATRA	1	0	0	0	0	0
MERCY	1	0	1	1	1	1
WORSER	1	0	1	1	1	0
...						

每篇文档表示成一个二值向量  $\in \{0, 1\}^M$

# 词频矩阵

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth ...
ANTHONY	157	73	0	0	0	1
BRUTUS	4	157	0	2	0	0
CAESAR	232	227	0	2	1	0
CALPURNIA	0	10	0	0	0	0
CLEOPATRA	57	0	0	0	0	0
MERCY	2	0	3	8	5	8
WORSER	2	0	1	1	1	5
...						

每篇文档表示成一个词频向量  $\in \mathbf{N}^M$

# 二值 $\rightarrow$ 词频 $\rightarrow$ 权重矩阵

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth ...
--	-----------------------------	------------------	----------------	--------	---------	----------------

ANTHONY	5.25	3.18	0.0	0.0	0.0	0.35
BRUTUS	1.21	6.10	0.0	1.0	0.0	0.0
CAESAR	8.59	2.54	0.0	1.51	0.25	0.0
CALPURNIA	0.0	1.54	0.0	0.0	0.0	0.0
CLEOPATRA	2.85	0.0	0.0	0.0	0.0	0.0
MERCY	1.51	0.0	1.90	0.12	5.25	0.88
WORSER	1.37	0.0	0.11	4.15	0.25	1.95
...						

每篇文档表示成一个基于**tf-idf**权重的实值向量  $\in \mathbf{R}^M$

# 文档表示成向量

- 每篇文档表示成一个基于tf-idf权重的实值向量  $\in \mathbf{R}^{|V|}$ .
- 于是，我们有一个  $|V|$ 维实值空间
- 空间的每一维都对应词项
- 文档都是该空间下的一个点或者向量
- 极高维向量：对于**Web**搜索引擎，空间会上千万维
- 对每个向量来说又非常稀疏，大部分都是**0**

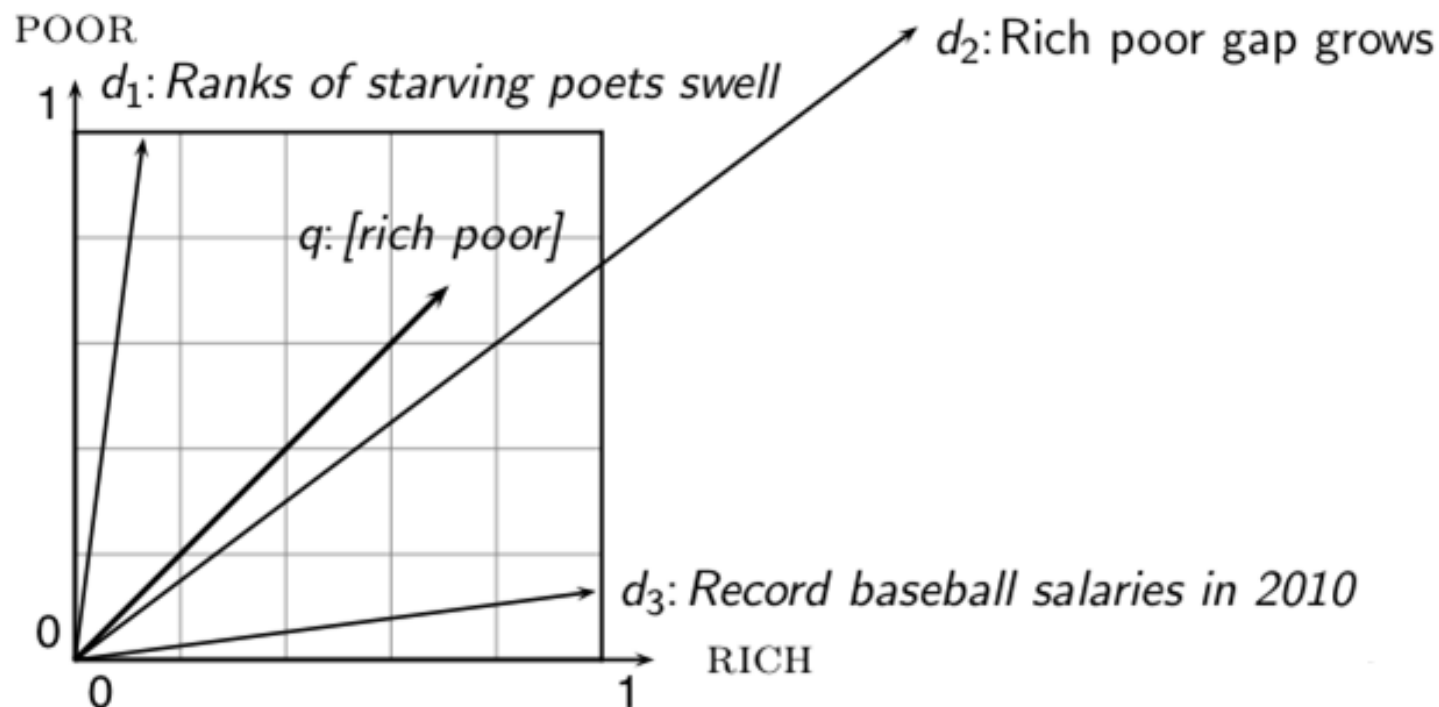
# 查询看成向量

- **关键思路1:** 对于查询做同样的处理，即将查询表示成同一高维空间的向量
- **关键思路2:** 按照文档对查询的邻近程度排序
  - 邻近度 = 相似度
  - 邻近度  $\approx$  距离的反面
- 回想一下，我们是希望和布尔模型不同，能够得到非二值的、既不是过多或也不是过少的检索结果
- 这里，我们通过计算出相关文档的相关度高于不相关文档相关度的方法来实现

# 向量空间下相似度的形式化定义

- 先考虑一下两个点之间的距离倒数
- 一种方法是采用欧氏距离
- 但是，欧氏距离不是一种好的选择，这是因为欧氏距离对向量长度很敏感

# 欧氏距离不好的例子



尽管查询  $q$  和文档  $d_2$  的词项分布非常相似，但是采用欧氏距离计算它们对应向量之间的距离非常大。

**Questions about basic vector space setup?**



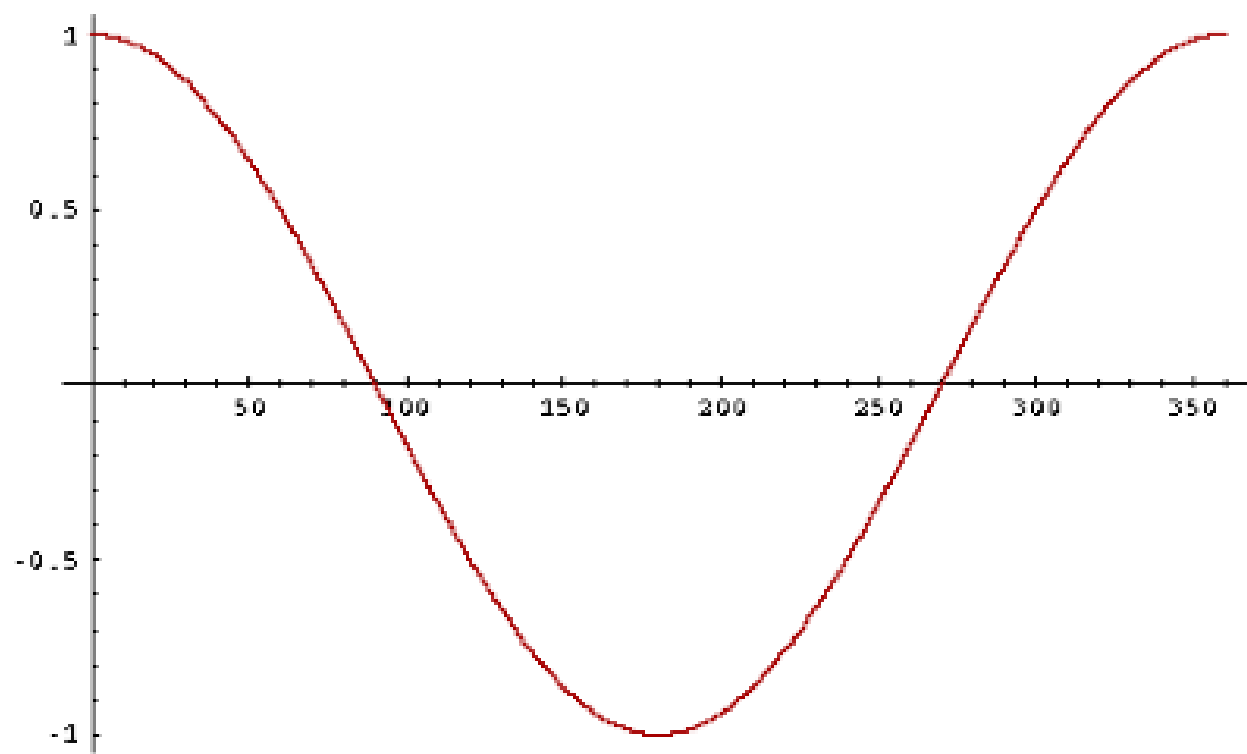
# 采用夹角而不是距离来计算

- 将文档按照其向量和查询向量的夹角大小来排序
- 假想实验：将文档  $d$  复制一份加在自身末尾得到文档  $d'$ .  
 $d'$  是  $d$  的两倍
- 很显然，从语义上看， $d$  和  $d'$  具有相同的内容
- 两者之间的夹角为  $0$ ，代表它们之间具有最大的相似度
- 但是，它们的欧氏距离可能会很大

# 从夹角到余弦

- 下面两个说法是等价的：
  - 按照夹角从小到大排列文档
  - 按照余弦从大到小排列文档
- 这是因为在区间 $[0^\circ, 180^\circ]$ 上，余弦函数**cosine**是一个单调递减函数

# Cosine函数

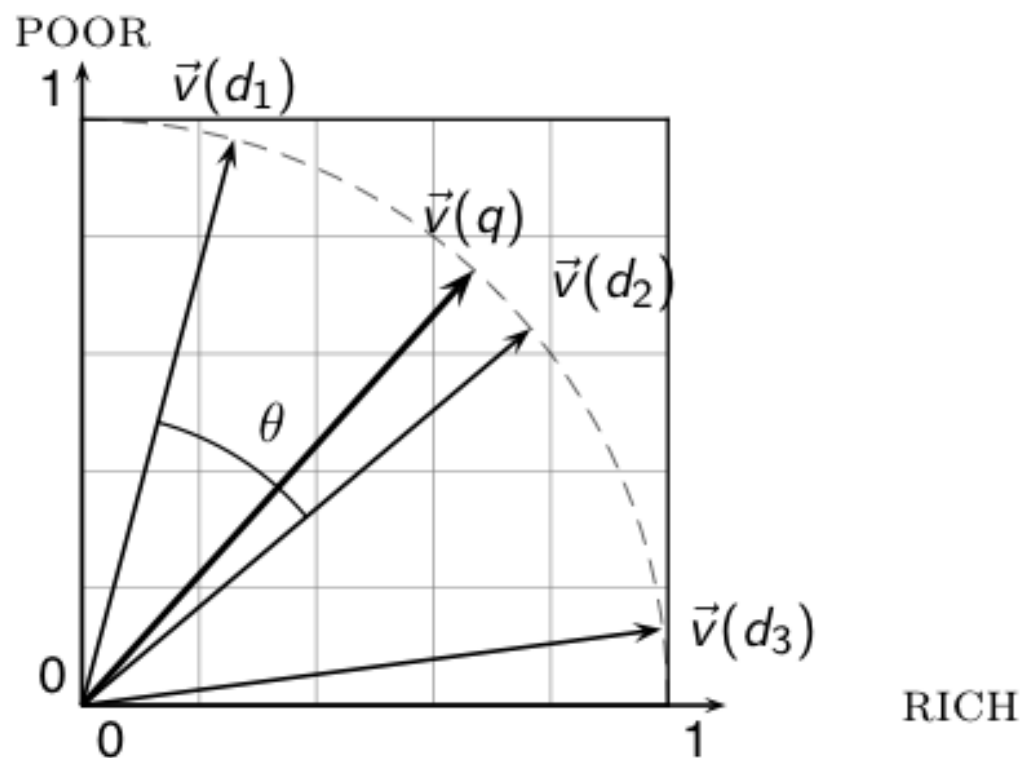


# 查询和文档之间的余弦相似度计算

$$\cos(\vec{q}, \vec{d}) = \text{SIM}(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}| |\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2} \sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

- $q_i$  是第  $i$  个词项在查询  $q$  中的 **tf-idf** 权重
- $d_i$  是第  $i$  个词项在文档  $d$  中的 **tf-idf** 权重
- $|\vec{q}|$  和  $|\vec{d}|$  分别是  $\vec{q}$  和  $\vec{d}$  的长度
- 上述公式就是  $\vec{q}$  和  $\vec{d}$  的余弦相似度，或者说向量  $\vec{q}$  和  $\vec{d}$  的夹角的余弦

# 余弦相似度的图示



# 余弦相似度的计算样例

## 3本小说之间的相似度

(1) **SaS(理智与情感): Sense and Sensibility**

(2) **PaP(傲慢与偏见): Pride and Prejudice**

(3) **WH(呼啸山庄): Wuthering Heights**

词项频率 $tf$

词项	SaS	PaP	WH
AFFECTION	115	58	20
JEALOUS	10	7	11
GOSSIP	2	0	6
WUTHERING	0	0	38

# 余弦相似度计算

词项频率 tf

词项	SaS	PaP	WH
AFFECTION	115	58	20
JEALOUS	10	7	11
GOSSIP	2	0	6
WUTHERING	0	0	38

对数词频 ( $1+\log_{10}tf$ )

词项	SaS	PaP	WH
AFFECTION	3.06	2.76	2.30
JEALOUS	2.0	1.85	2.04
GOSSIP	1.30	0	1.78
WUTHERING	0	0	2.58

为了简化计算，上述计算过程中没有引入idf

# 余弦相似度计算

对数词频( $1 + \log_{10} \text{tf}$ )

数词频的余弦归一化结果

词项	SaS	PaP	WH
AFFECTION	3.06	2.76	2.30
JEALOUS	2.0	1.85	2.04
GOSSIP	1.30	0	1.78
WUTHERING	0	0	2.58

词项	SaS	PaP	WH
AFFECTION	0.789	0.832	0.524
JEALOUS	0.515	0.555	0.465
GOSSIP	0.335	0.0	0.405
WUTHERING	0.0	0.0	0.588

$\cos(\text{SaS}, \text{PaP}) \approx 0.789 * 0.832 + 0.515 * 0.555 + 0.335 * 0.0 + 0.0 * 0.0 \approx 0.94.$

$\cos(\text{SaS}, \text{WH}) \approx 0.79$

$\cos(\text{PaP}, \text{WH}) \approx 0.69$

$\cos(\text{SaS}, \text{PaP}) > \cos(\text{SAS}, \text{WH}) > \cos(\text{PaP}, \text{WH})$



# 余弦相似度计算算法

COSINESCORE( $q$ )

```
1  float Scores[ $N$ ] = 0
2  float Length[ $N$ ]
3  for each query term  $t$ 
4  do calculate  $w_{t,q}$  and fetch postings list for  $t$ 
5      for each pair( $d, tf_{t,d}$ ) in postings list
6      do Scores[ $d$ ] + =  $w_{t,d} \times w_{t,q}$ 
7  Read the array Length
8  for each  $d$ 
9  do Scores[ $d$ ] = Scores[ $d$ ] / Length[ $d$ ]
10 return Top  $K$  components of Scores[]
```

# 向量空间模型小结

- 将查询表示成 **tf-idf** 权重向量
- 将每篇文档表示成同一空间下的 **tf-idf** 权重向量
- 计算两个向量之间的某种相似度(如余弦相似度)
- 按照相似度大小将文档排序
- 将前  **$K$**  (如  **$K=10$** ) 篇文档返回给用户

# GERARD SALTON(1927-1995)

信息检索领域的奠基人之一，向量空间模型的完善者和倡导者，**SMART**系统的主要研制者，**ACM Fellow**

**1958**年毕业于哈佛大学应用数学专业，是**Howard Aiken**的关门博士生。

**Howard Aiken**是**IBM**第一台大型机**ASCC**的研制负责人。

是康奈尔大学计算机系的创建者之一。



**LUCENE**

# SUMMARY

- **Apache Lucene是一个基于Java全文搜索引擎，利用它可以轻易地为Java软件加入全文搜寻功能。**
- **Lucene不是一个完整的搜索应用程序，而是一个基于 Java 的全文信息检索工具包，为你的应用程序提供索引和搜索功能，可以方便的嵌入到各种应用中实现针对应用的全文索引/检索功能。**
- **Lucene 目前是 Apache Jakarta 家族中的一个开源项目。也是目前最为流行的基于 Java 开源全文检索工具包。**

# HISTORY



## 贡献者：

Doug Cutting是一位资深全文索引/检索专家，曾经是V-Twin搜索引擎(Apple的Copland操作系统的成就之一)的主要开发者。作为Lucene和Nutch两大Apach Open Source Project的始创人(其实还有Lucy, Lucene4C 和Hadoop等相关子项目)，Doug Cutting 一直为搜索引擎的开发人员所关注。他终于在为Yahoo以Contractor的身份工作4年后，于06年正式以Employee的身份加入Yahoo。他贡献出的Lucene的目标是为各种中小型应用程序加入全文检索功能。

## 发展历程：

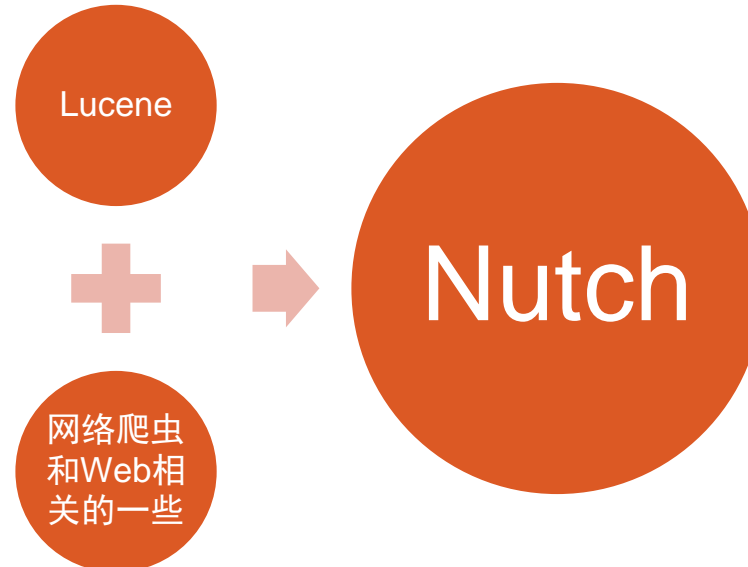
最先发布在作者自己的www.lucene.com，后来发布在Source Forge，2001年年底成为APACHE基金会jakarta的一个子项目：

<http://jakarta.apache.org/lucene/>

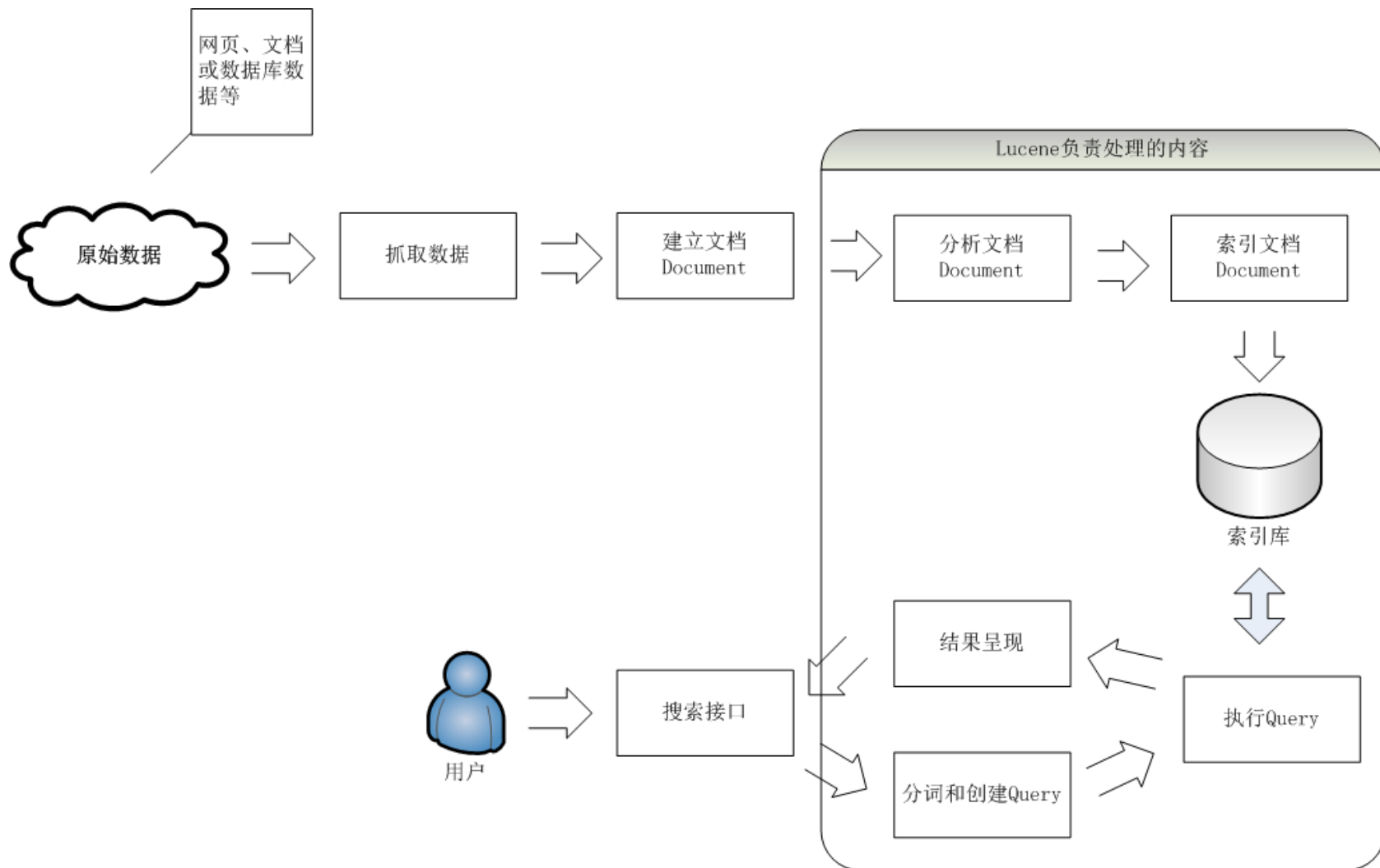
# TIP:LUCENE VS NUTCH

Lucene是一个提供全文文本搜索的函数库，它不是一个应用软件。它提供很多API函数让你可以运用到各种实际应用程序中。

Nutch是一个建立在Lucene核心之上的Web搜索的实现，它是一个真正的应用程序。



# LUCENE全文搜索处理流程





# LUCENE的创新

	Lucene	其他开源全文检索系统
增量索引和批量索引	可以进行 <b>增量的索引</b> (Append), 可以对于大量数据进行批量索引, 并且接口设计用于优化批量索引和小批量的增量索引。	很多系统只支持批量的索引, 有时数据源有一点增加也需要重建索引。
数据源	Lucene没有定义具体的数据源, 而是一个文档的结构, 因此可以 <b>非常灵活的适应各种应用</b> (只要前端有合适的转换器把数据源转换成相应结构),	很多系统只针对网页, 缺乏其他格式文档的灵活性。
索引内容抓取	Lucene的文档是由 <b>多个字段</b> 组成的, 甚至可以控制那些字段需要进行索引, 那些字段不需要索引, 进一步索引的字段也分为需要分词和不需要分词的类型: 需要进行分词的索引, 比如: 标题, 文章内容字段 不需要进行分词的索引, 比如: 作者/日期字段	缺乏通用性, 往往将文档整个索引了

# LUCENE的创新

语言分析	通过 <b>语言分析器的不同扩展实现</b> ： 可以过滤掉不需要的词：an the of 等， 西文语法分析：将jumps jumped jumper都归结成jump进行索引/检索 非英文支持：对亚洲语言，阿拉伯语言的索引支持	缺乏通用接口实现
查询分析	通过查询分析接口的实现，可以 <b>定制自己的查询语法规则</b> ： 比如：多个关键词之间的 + - and or 关系等	
并发访问	能够 <b>支持多用户</b> 的使用	