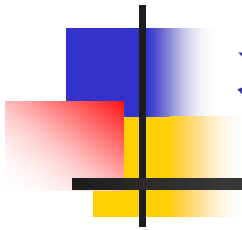
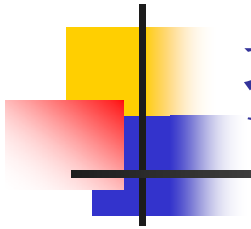


搜索引擎技术

刘挺
哈工大信息检索研究室
2004年秋

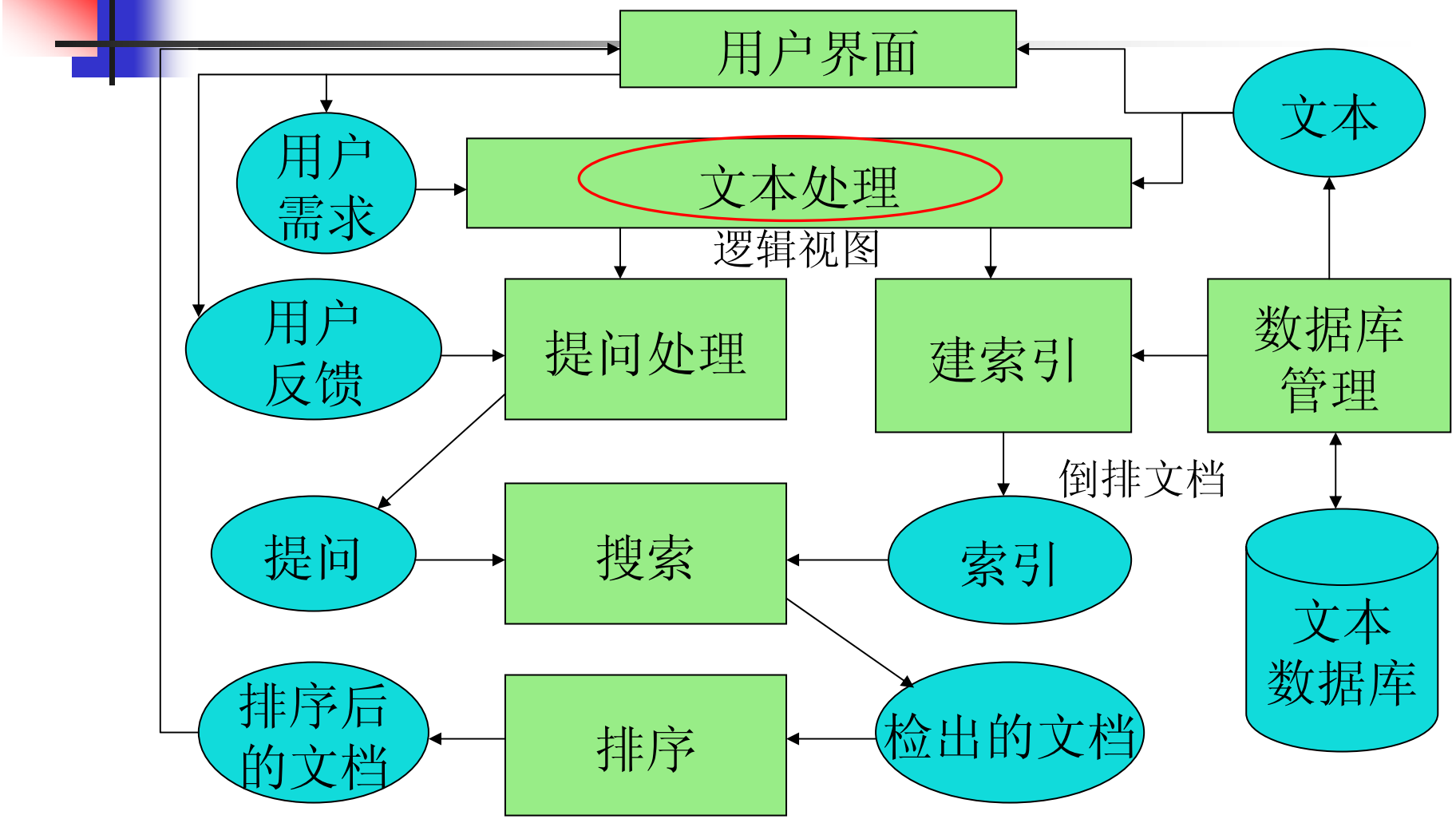


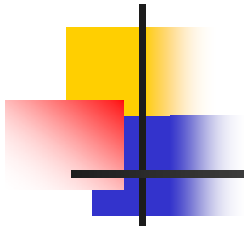
提纲

- 文本处理
  - term处理
  - 文本的特性
- 索引与检索
  - 倒排文件
  - Signature文件
  - PAT Tree
- Query处理
  - 相关反馈
  - 查询扩展

文本处理
——term处理

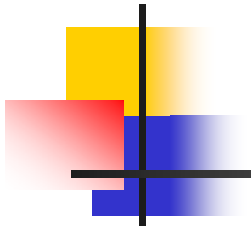
信息检索系统的体系结构





文本表示

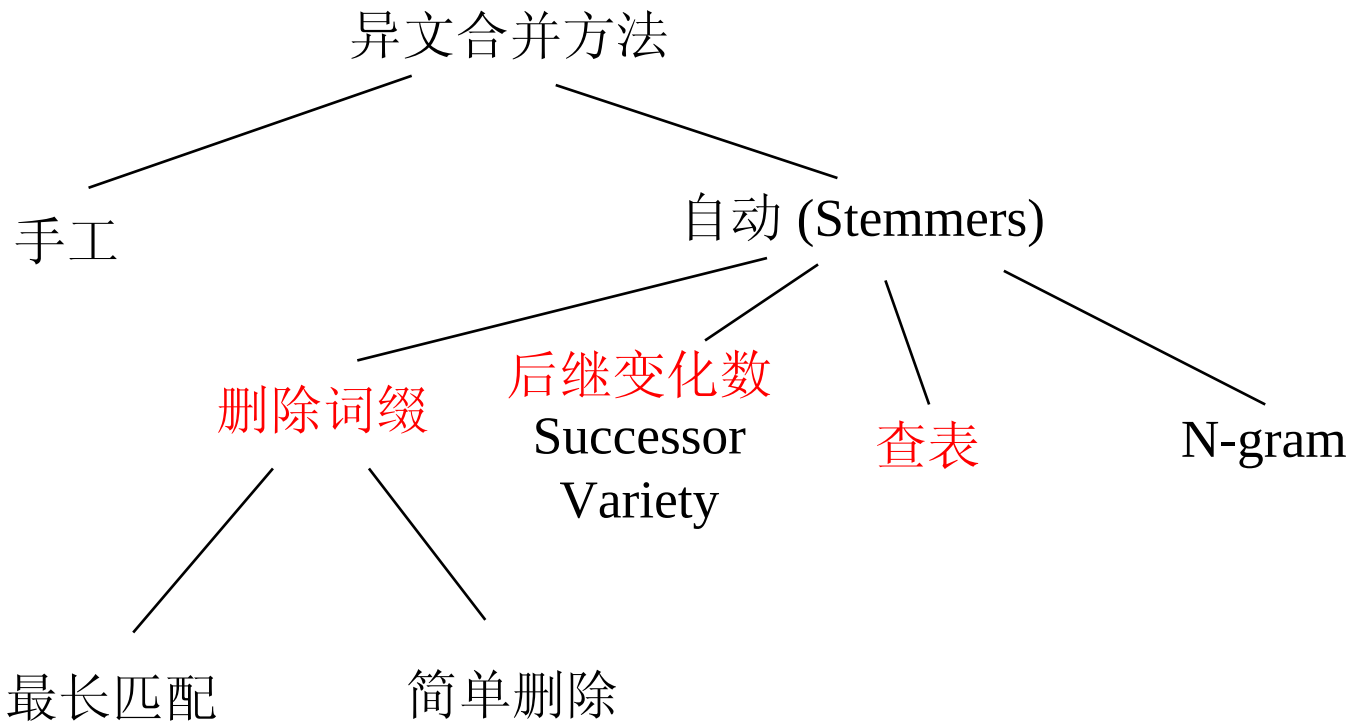
- 文本可以表示为
  - 一个字符串
  - 词的集合
  - 语言单元 (例如：名词、短语)
- 简单的表示 (如：单个词项) 效果好
  - 以往的一些研究显示：基于短语的索引不如基于词的索引
  - 短语可能太特殊了

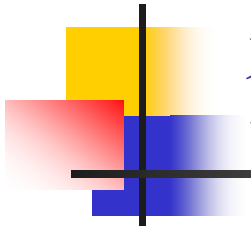


Stemming

- 克服词形的变化，把所有同根词转变为单一形式
  - *RECOGNIZE, RECOGNISE, RECOGNIZED, RECOGNIZATION*
- Stemming的优点:
  - 减少不同term的数量
  - 识别相似的词
  - 改进了检索性能，但不采用语言分析的方法
- Stemming的缺点：
  - 正确率显然达不到100%
  - 不正确的stemming算法可能改变词的含义
  - 需要避免过分的截断
    - MEDICAL和MEDIA被识别为MED*，并被认为是意义相近的，这就错了

异文合并(Conflation)方法



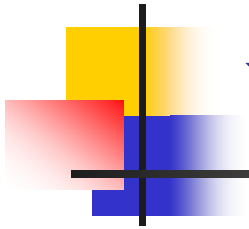


查表

- 创建一个term和stem的对应表

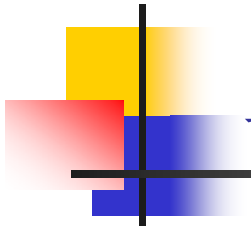
TERM	STEM
engineering	engineer
engineered	engineer
engineer	engineer

- 表可以被索引起来，以便加快查找速度
- 创建这样的表很困难
- 存储空间的开销较大



词缀删除算法

- 词缀删除算法将term的前缀和/或后缀删除，留下词干
- 大多数算法删除后缀，例如：-SES, -ATION, -ING等等
  - 最长匹配
    - 从词中删除最长匹配的后缀:
      computability --> comput
      singing --> sing
      avoid: *ability ->NULL, sing->s*
  - 迭代式最长匹配
    - 重复最长匹配的过程：
    - WILLINGNESS --> 删除NESS --> 删除ING

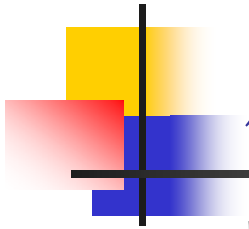


上下文有关和上下文无关

- 上下文无关
  - 根据后缀表删除后缀(或基于规则集)
- 上下文有关
  - 考虑词的其它性质，例如：
    - *happily* → *happi* → *happy*
    - 定义一个上下文敏感的转换规则：如果一个词根以i结尾，i前面是p，那么将i转换为y
- 需要控制许多例外规则
  - 从TABLE中删除-ABLE不行，从GAS中删除-S也不行
  - 有时需要删除“双写字母”
    - FORGETTING → FORGET

Porter算法 (1980)

- 每一步有一组上下文无关或有关的规则用来删除后缀，或者将其转换为其它形式
  - 上下文无关规则：sses → ss, ies → i, s → NULL
  - 上下文有关规则：
    (*v*) : ed → NULL, ing → NULL
    (*v*)的含义是：词根必须包含一个元音
    plastered → plaster
    bled → bled　删除词缀后，剩下的词干里没有元音
- 问题：
  - 需要大量的语言知识来定义规则
  - 由于人类语言的复杂性，规则无法覆盖全部情况
  - 规则依赖于语言



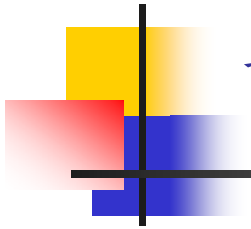
后继变化数Successor Variety

- 基于对文本集合的统计分析
  - 给定一个足够大的语料库, 可以通过统计的方法获得词干
  - 这种方法是自动的，和语言关联性不大的
- 后继变化数的定义：
  - 语料库中跟在某个字符串后的不同字符的数
- 考虑英文词典
  - pr? -> 后继变化数是多少？
  - pro? -> ？
  - *pr* 和 *pro* 谁更像一个词根?
  - 直觉：如果一个字符串的后继变化数值很低，则可能是一个词根

后继变化数的例子

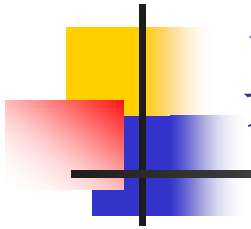
Corpus
ABLE, APE
BEATABLE
FIXABLE
READ
READABLE
READING
READS
RED
ROPE, RIPE

Prefix	Successor Variety	Letters
R	3	E,I,O
RE	2	A,D
REA	1	D
READ	3	A,I,S
READA	1	B
READAB	1	L
READABL	1	E
READABLE	1	BLANK



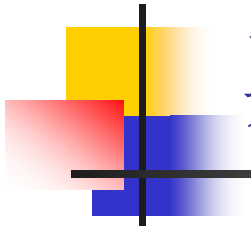
切分

- 使用后继变化数信息切分词
- cut off
  - 通过后继变化数的cutoff值识别边界
  - 当后继变化数>=阈值时，进行切分
  - 考虑阈值 = 2 R|E|AD|ABLE
- 尖峰和高地
  - 在后继变化数比前后都大，出现尖峰的位置切开
    - READ|ABLE
- 切出来的词必须完整
  - 如：READ



其它term处理

- 英文词形态还原
  - Calculated, Calculating -> Calculate
  - Went->go, goes->go
- 中文分词
  - 举例
    - “他将来北京”
    - “研究*生命*的起源”
  - 英文有没有分词问题？
    - 有，例如“give in”（投降），必须分词



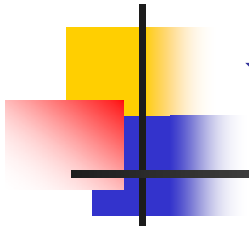
其它term处理（续）

- 词性标注
  - 采用HMM的算法
  - 参加下页词性标记表
- 词义消歧
  - 采用贝叶斯等算法
  - 《同义词词林》，参见后文
- 停用词表
  - 没有固定的标准
  - 英文：the, a, and, ……
  - 中文：的，了，和， ……

词性标注表

普通名词：n	时间名词：nt	方位名词：nd	处所名词：nl	人名：nh	地名：ns
团体、机构、组织的专名：ni	其它专名：nz	动词：v	形容词：a	区别词：b	副词：d
数词：m	量词：q	代词；r	介词：p	连词：c	叹词：e
拟声词：o	助词：u	前接成分：h	后接成分：k	习用语：i	简称：j
语素字：g	非语素字：x	标点：wp	字符串：ws		

文本处理
——文本的特性



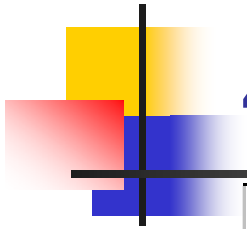
词频

- 不同词的频率是怎样分布的？
- 极少的词是非常常见的
  - 英文中最常用的两个词是： “the”, “of”，他们的出现频率占全部英文词的10%
- 大多数词很少出现
  - 语料库中的一半词只出现一次
  - 称为“heavy tailed”分布, 因为大多数的概率值都是 “tail”

样本次品数据
(from B. Croft, UMass)

Frequent Word	Number of Occurrences	Percentage of Total
the	7,398,934	5.9
of	3,893,790	3.1
to	3,364,653	2.7
and	3,320,687	2.6
in	2,311,785	1.8
is	1,559,147	1.2
for	1,313,561	1.0
The	1,144,860	0.9
that	1,066,503	0.8
said	1,027,713	0.8

Frequencies from 336,310 documents in the 1GB TREC Volume 3 Corpus
125,720,891 total word occurrences; 508,209 unique words



Zipf's Law

Rank(R)	Term	Frequency (F)	R*F (10**6)
1	the	69,971	0.070
2	of	36,411	0.073
3	and	28,852	0.086
4	to	26,149	0.104
5	a	23,237	0.116
6	in	21,341	0.128
7	that	10,595	0.074
8	is	10,009	0.081
9	was	9,816	0.088
10	he	9,543	0.095

Zipf(齐普夫)'s定律

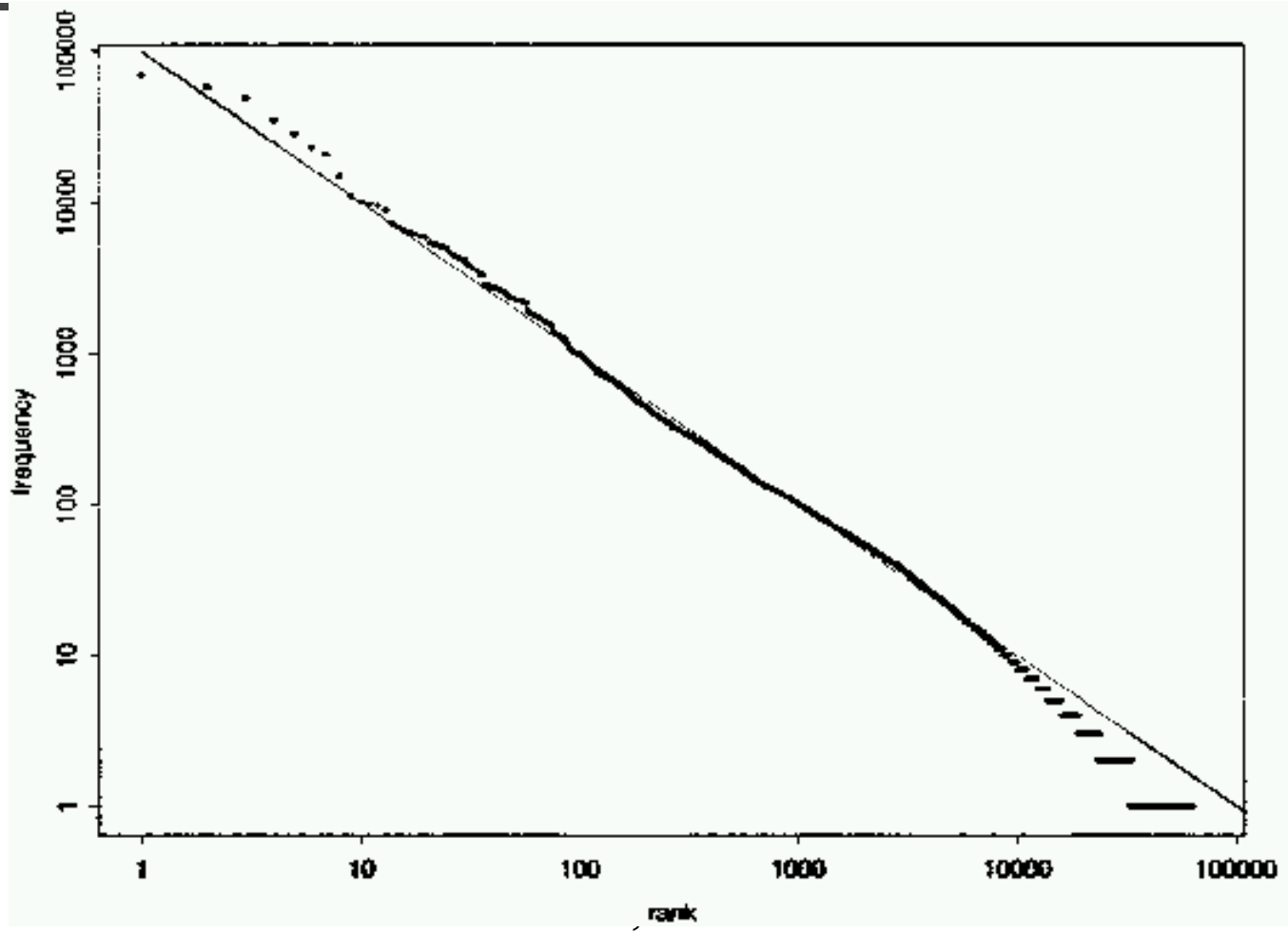
- Rank (*r*): 在按词频(*f*)降序排列的词表中所处的位置.
- Zipf (1949) "发现":

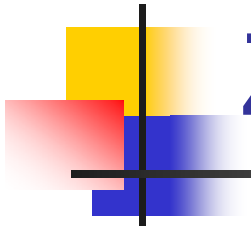
$$f \propto \frac{1}{r} \qquad f \cdot r = k \ \text{(for constant } k\text{)}$$

- 如果rank为*r*的词的概率位p_r，*N*是所有词出现的次数:

$$p_r = \frac{f}{N} = \frac{A}{r} \quad \text{for corpus indp. const. } A \approx 0.1$$

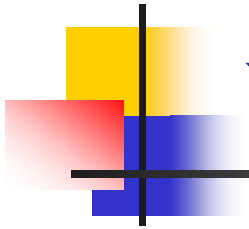
Brown语料库验证Zipf定律





Zipf定律对IR的影响

- 好消息:
  - 停用词占文本中的很大一部分，因此删除停用词可以大量减少倒排文档的存储空间
- 坏消息:
  - 对大多数词来说，进行词汇之间的相关分析并不容易，因为它们出现的比较少



词表增长

- 随着语料库的增长，词表以什么样的速度相应地增长？
- 这决定了随着语料库规模的增长，倒排文件需要怎样增长
- 由于专名的存在，词表实际上没有上限

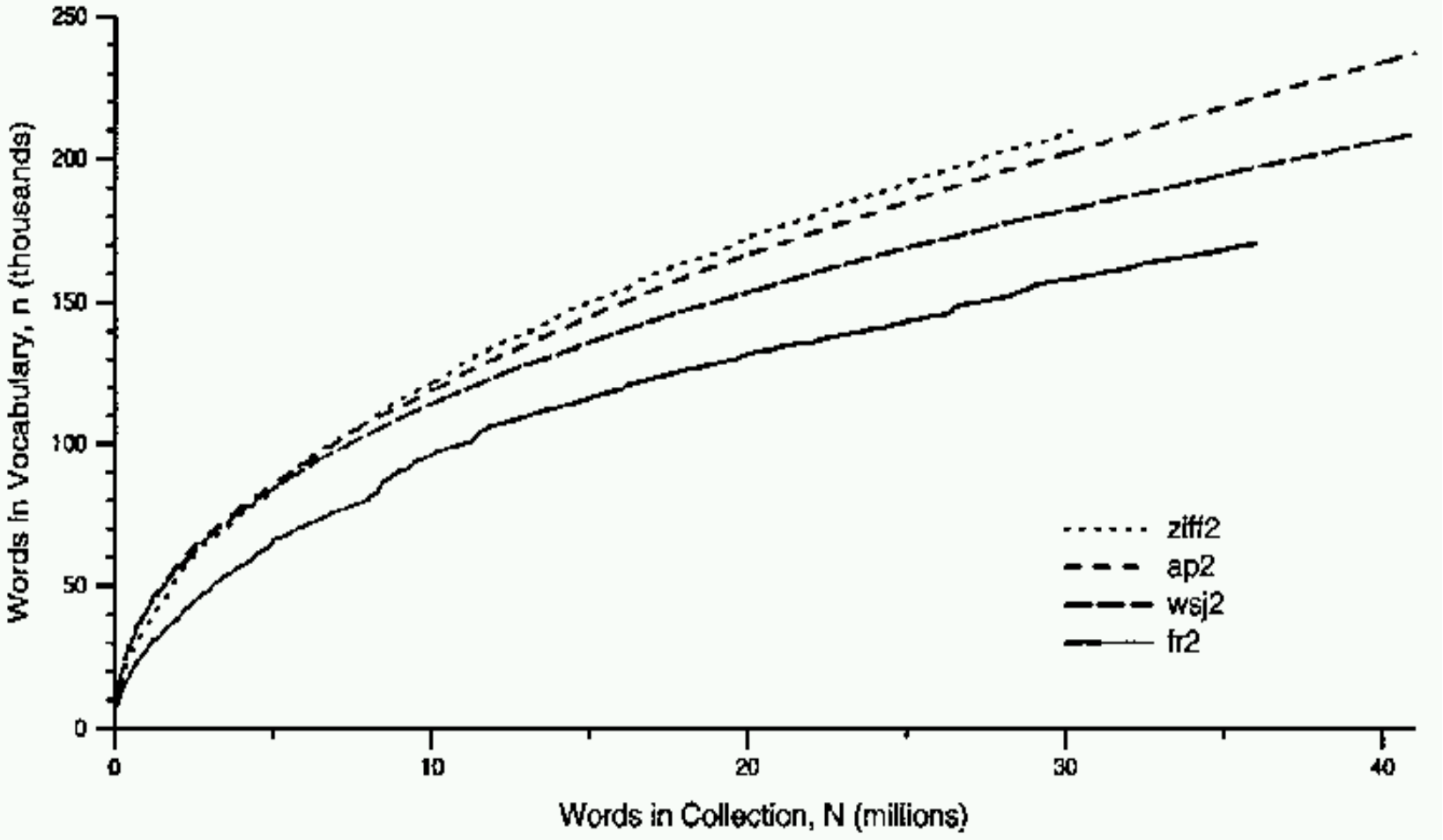
Heaps' Law

- V 是词表大小，n 是语料库的长度（词书）

  $V = Kn^{\beta}$ with constants K, $0 < \beta < 1$

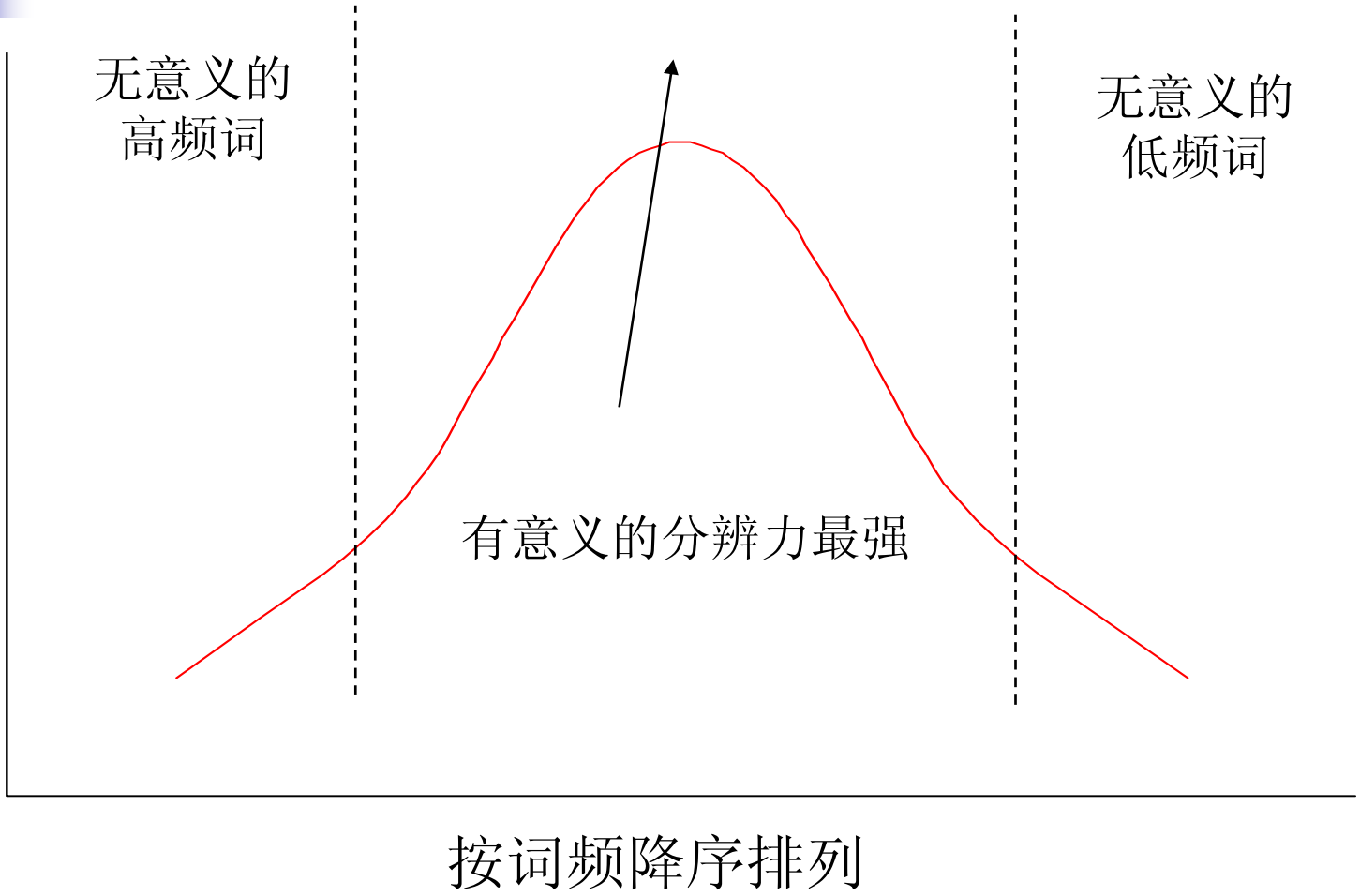
- 典型的常数:
  - $K \approx$ 10–100
  - $\beta \approx$ 0.4–0.6 (approx. square-root)

Heaps’ 定律


Words in Vocabulary, n (thousands)
250
200
150
100
50
0
0
10
20
30
40
Words in Collection, N (millions)
ziff2
ap2
wsj2
fr2

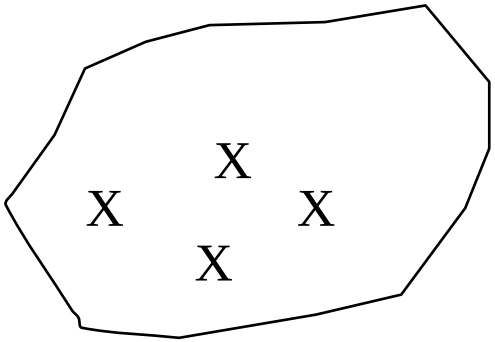
词的分辨力

分辨力是一个词作为特征将它所在的文档与其它文档区别开来的能力

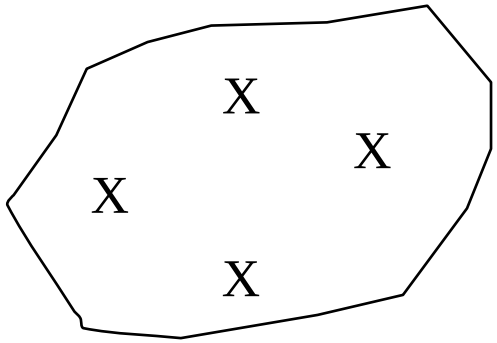


索引项的分辨力

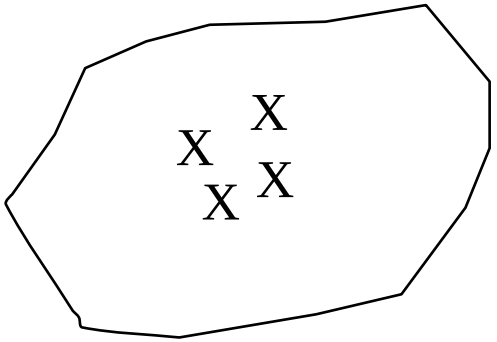
- 好的索引项能够将文档尽可能地离散开
  - 例如：在一个关于“计算机科学”的文档集合中



原始文档空间：
system



添加了好的索引项：
system, database



添加了不好的索引项：
system, computer

索引项分辨力举例

	all terms	indexed	bad	good
d1	(a,b,c,d,r)	(b,c,d)	(a,b,c,d)	(b,c,d,r)
d2	(a,b,n,d,r)	(b,n,d)	(a,b,n,d)	(b,n,d,r)
d3	(a,m,p,q)	(m,p,q)	(a,m,p,q)	(m,p,q)
d4	(a,x,p,q)	(x,p,q)	(a,x,p,q)	(x,p,q)

- a就不是一个好的索引项，因为各个文档都包含a
- r可以使*d1*和*d2*靠近，并使它们远离 *d3*和*d4*

索引项分辨力的计算

- 好的索引空间能够使文档集中各文档对之间的相似度的总和尽可能小
- N各文档之间的平均相似度为：

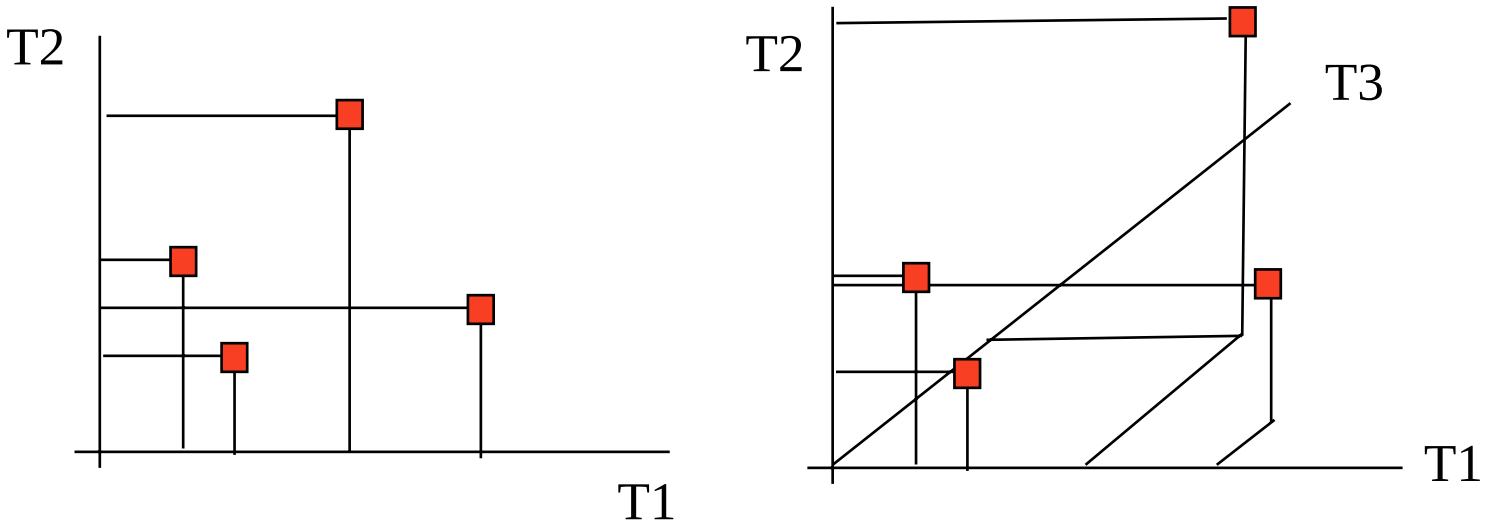
$$Q = \frac{1}{N(N-1)} \sum_{i=1}^{N} \sum_{\substack{k=1 \\ k \neq i}}^{N} sim(D_i, D_k)$$

- 假如Q和 Q_j是加入term j之前和之后的平均相似度，这term j的分辨力为

$$\boxed{dv_j = Q - Q_j}$$

索引项分辨力（图示）

- 加一个索引项就加了一维空间，这将使包含该索引项的文档和不包含该索引项的文档拉开距离



文本质心

- dv_j 的计算开销太大
  - 需要针对每个term，计算 $N(N-1)/2$ 各文档对的相似度
  - 文档更新后，需要重新计算
- 这种方案:
  - 找到文档空间的质心(centroid)

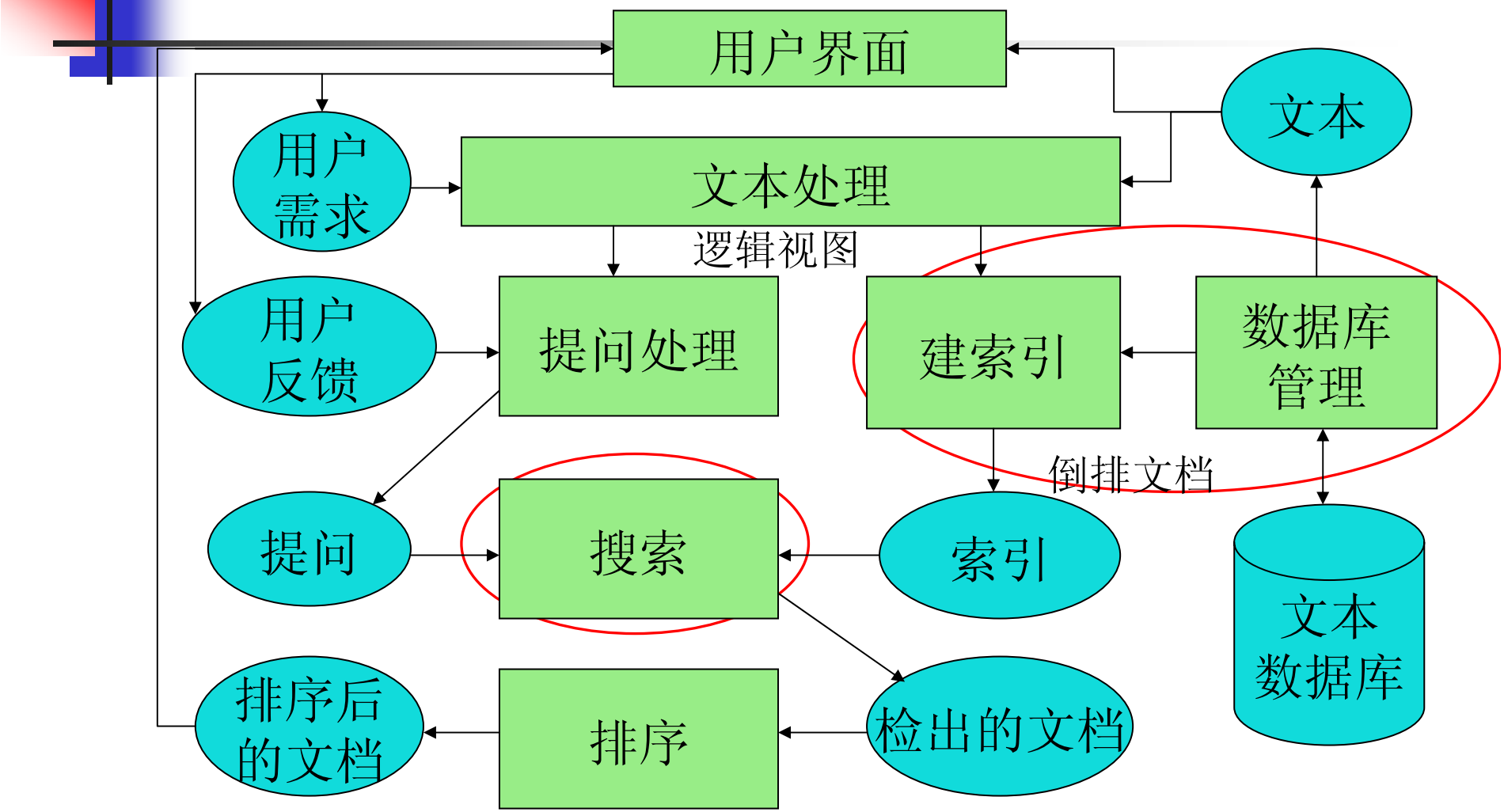
$$C = (c_1, c_2, \ldots, c_t), \qquad c_j = \frac{1}{N}\sum_{k=1}^{N} d_{kj}$$

  - Q是每个文档和质心之间的相似度

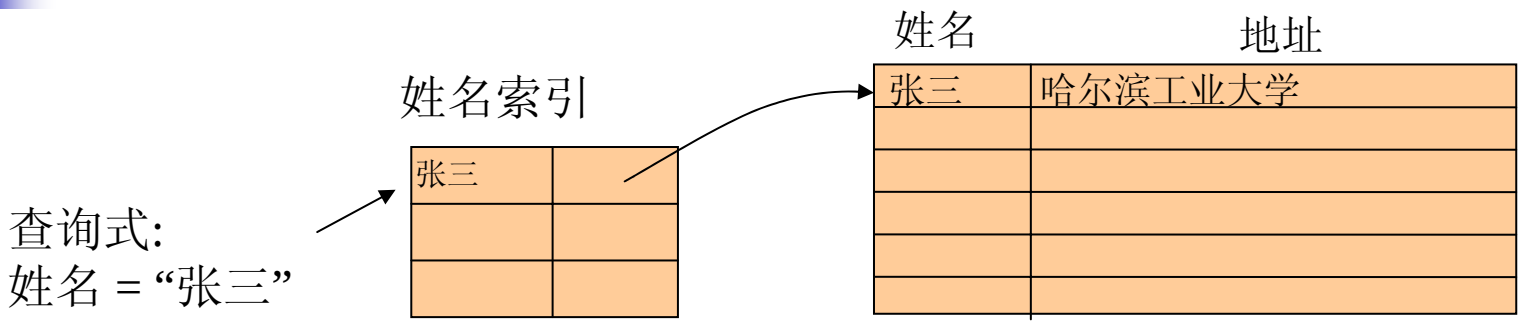
$$Q = \frac{1}{N}\sum_{k=1}^{N} sim(C, D_k)$$

索引与检索
——倒排文档

信息检索系统的体系结构

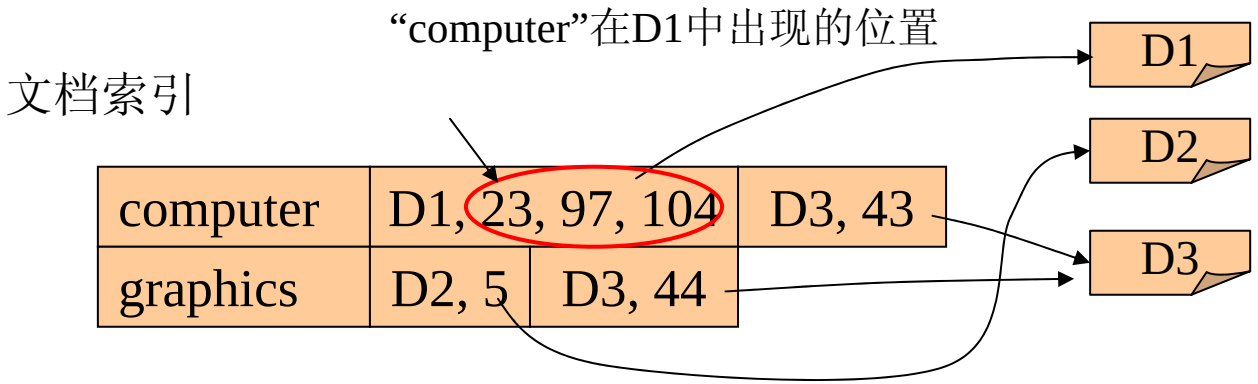


在关系数据库上建索引

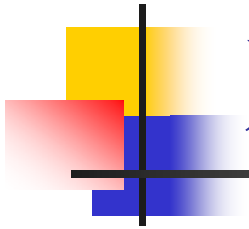


- 索引结构: hashing, B+-tree
- 索引全部记录，在全部记录上进行搜索
- 只有有用的字段被索引
- 精确地快速地查找

对文档进行索引



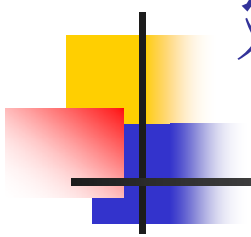
- 索引结构: ***hashing, B+-trees, tries***.
- 部分匹配: ’%comput% ’
- 短语搜索:查找包含“computer graphics”的文档



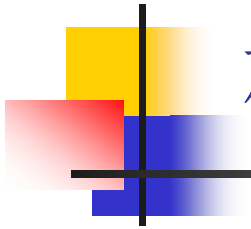
建立索引的过程

- 识别文档中的词
- 删除停用词(stop words)
- 提取词干(stemming)
- 用索引项的标号代替词干(stems)
- 统计词干的数量(*tf*)
- (可选) 对低频词项使用同义词词典(thesaurus)
- (可选) 对高频词项构成短语
- 计算所有单个词项、短语和语义类的权重

建立索引的过程 – 举例



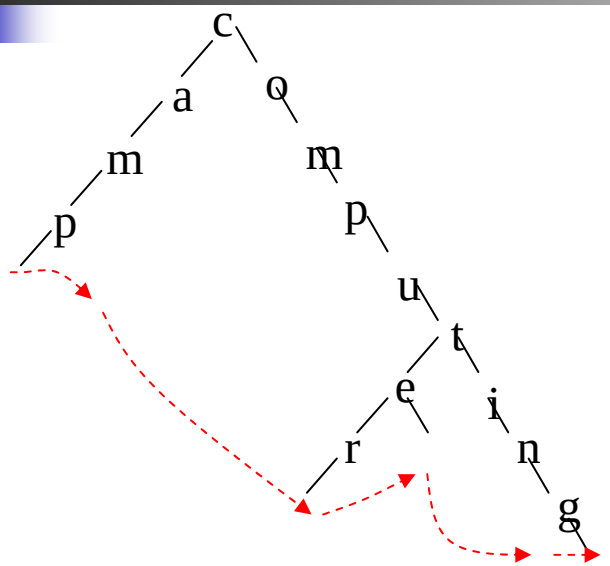
- 输入文本
  - *The analysis of 25 indexing algorithms has not produced consistent retrieval performance. The best indexing technique for retrieving documents is not known*
- 删除**stopwords**
  - *analysis indexing algorithms produced consistent retrieval performance best indexing technique retrieving documents known*
- **Stemming**
  - *analysis index algorithm produc consistent retriev perform best index technique retriev document known*
- 转换为索引编号
  - 123 345 110 2234 432 3565 2302 566 345 4321 3565 755 1128
- 计算***tf***
  - 110 1 123 1 345 2 1 432 1 566 1 755 1 1128 1 2302 1 2344 1 3565 2 4321 1
- 计算词项的权值 **(**依赖于使用的模型**)**



检索过程

- 给定query
- 对query进行stemming，算法与对文档的处理相同
- 用索引编号代替stems
- 计算所有query terms的权重
- 形成query向量（对VSM模型而言）
- 计算query向量和文档向量之间的相似度
- 返回排序后的文档集合

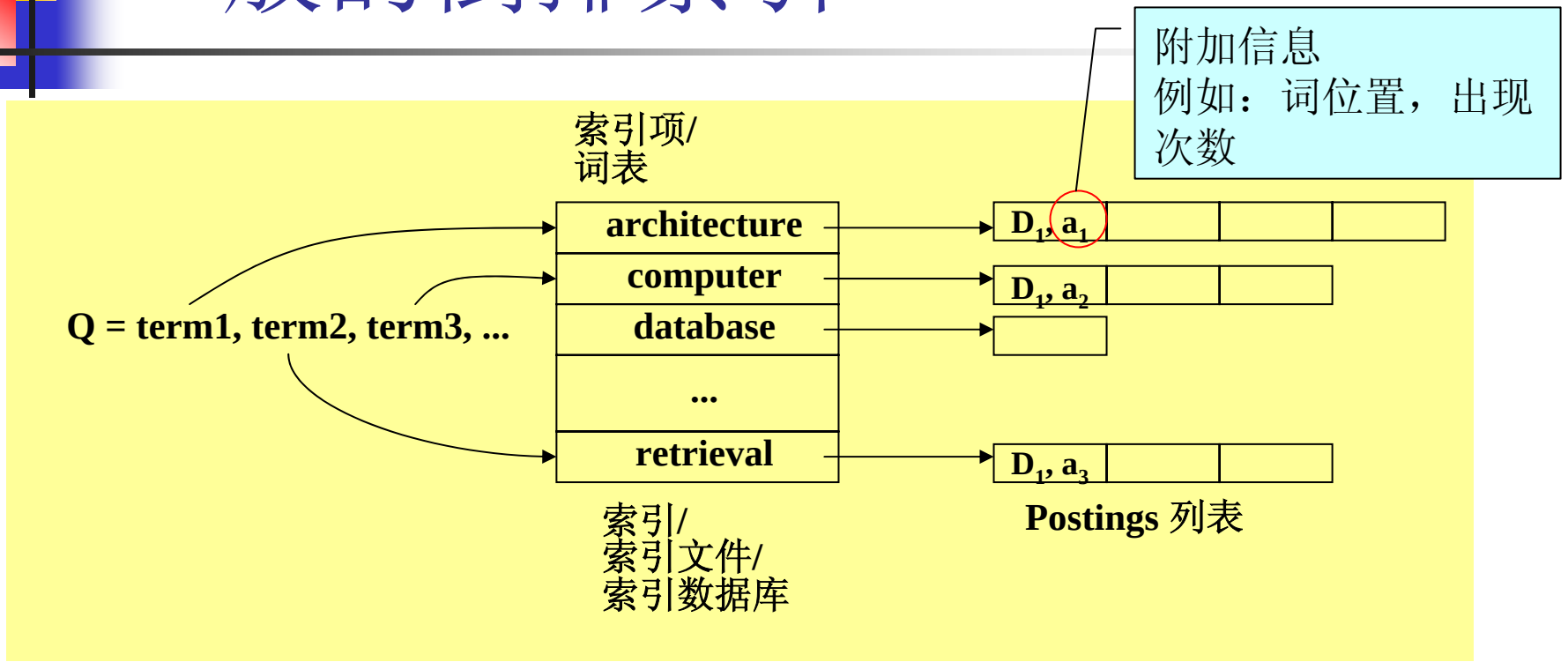
Trie



- 对词内的部分匹配和一个词集的顺序检索效果好
- Trie的大小依赖于词间重叠的数量

- B-tree/trie对关键词的顺序检索效果好，例如：“找出所有按字母顺序看在camp和computing之间”
- Hash文件对于单个词的快速随机检索效果好

一般的倒排索引

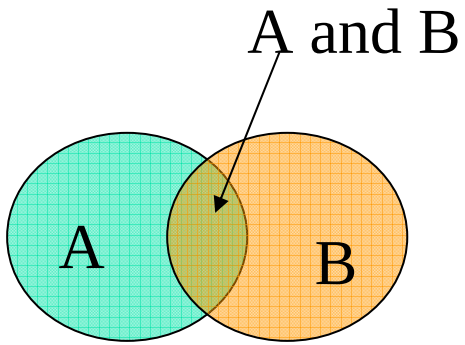


- 索引文件可以用任何文件结构来实现
- 索引文件中的词项是文档集合中的词表

倒排索引上的布尔检索

- 一个布尔检索包含***n***个用布尔操作连接的词项，例如："computer AND news AND NOT newsgroup"
  - 可以用括号来调整逻辑运算次序

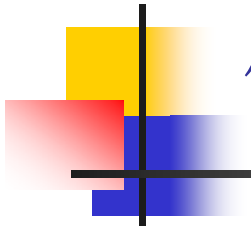
- 每个term从倒排索引中返回一个postings list
  - 如果term不在任何文档中出现，则postings list为空
- 检索结果根据逻辑关系相结合：
  - AND: 集合做交运算
  - OR: 集合做并运算
  - NOT: 集合做差运算



倒排索引上的布尔检索

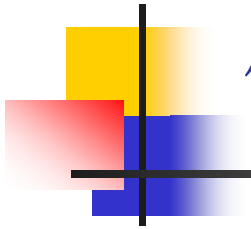
- 标准的优化技术应用：
  - 从最短的posting list开始做“与”操作，保证中间结果越小越好
  - “网络” AND “病毒” AND “蠕虫”
  - 从哪个词项开始做交运算呢？

显然是：“病毒”和“蠕虫”



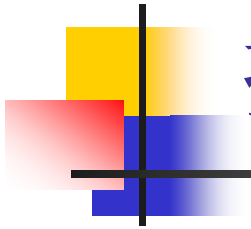
倒排索引的优点

- 快速索引 (长query需要更多时间)
- 灵活性: 不同类型的信息都可以存储在postings list中
- 如果存储了足够多的信息，则可以支持复杂的检索操作
  - 例如：如果记录了词在文档中的准确位置，就可以支持短语检索，或模糊检索



倒排索引的缺点

- 很大的存储开销
  - 50% - 150% - 300%
- 更新、插入和删除都需要很高的维护开销，倒排索引相对静态的环境(很少插入和更新)中使用比较好
- 处理开销随着布尔操作的增加而增长
- 由于postings越来越多(例如引入同义词)，导致索引检索的代价越来越大，需要对位置进行很多处理(例如短语匹配)

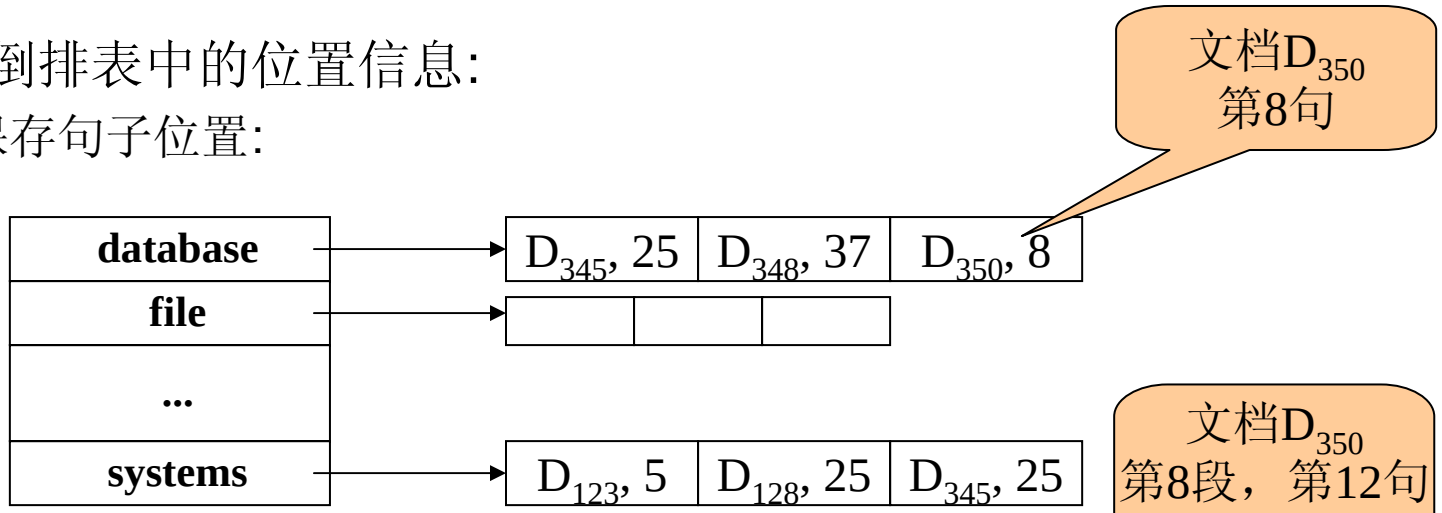


扩展 – 距离约束

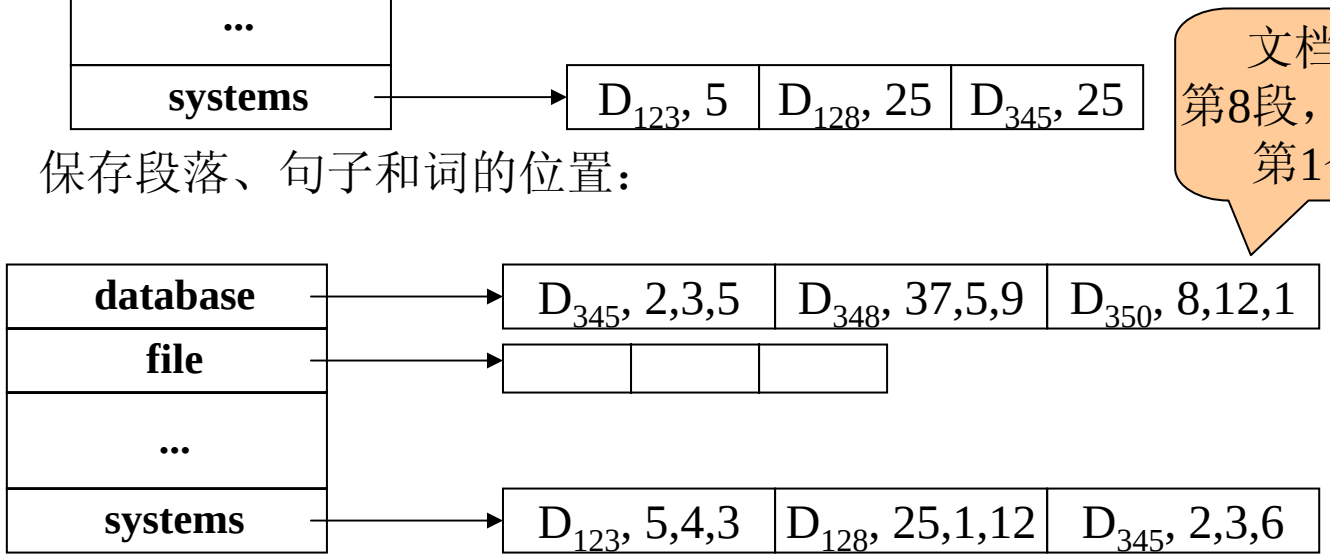
- 常常需要知道邻接条件，例如：
  - “database” 后面紧跟着“systems”
    - 例如：短语搜索 “database systems”
  - “database”和“systems”之间不能间隔超过3个词
  - “database”和“architecture”在同一个句子里
- 需求扩展:
  - 倒排索引中保存着关键词在文档中的位置，文档的组成单元(标题, 小标题, 句子分割标记等)
  - 检索算法和位置信息相关联，并需检查文档的组成单元

扩展 – 距离约束

- 保存倒排表中的位置信息:
  - 保存句子位置:

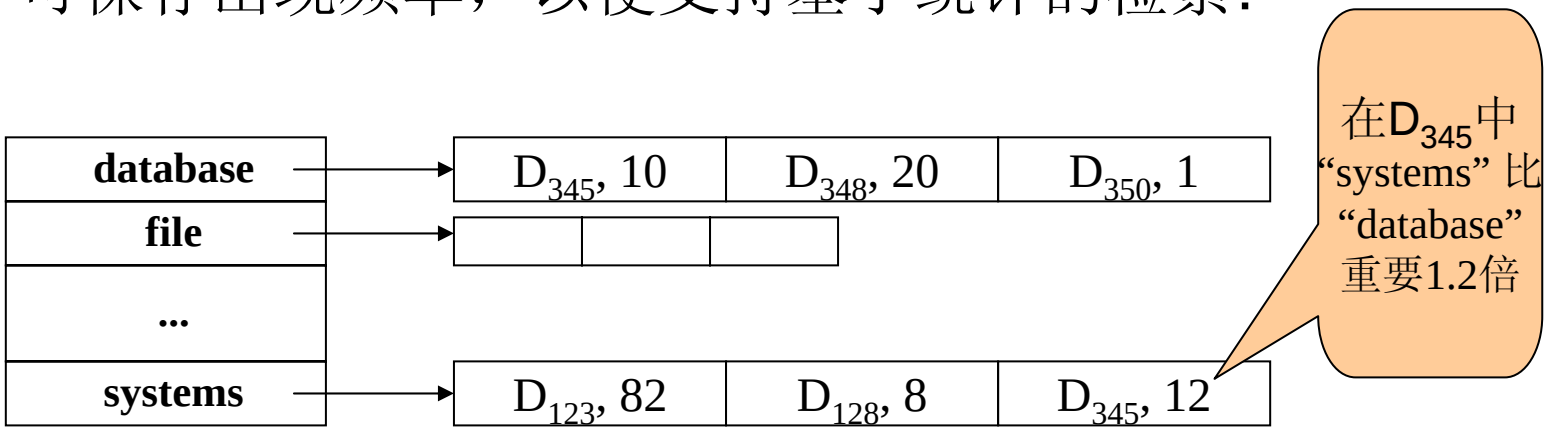


  - 保存段落、句子和词的位置:



扩展 – Term的权重

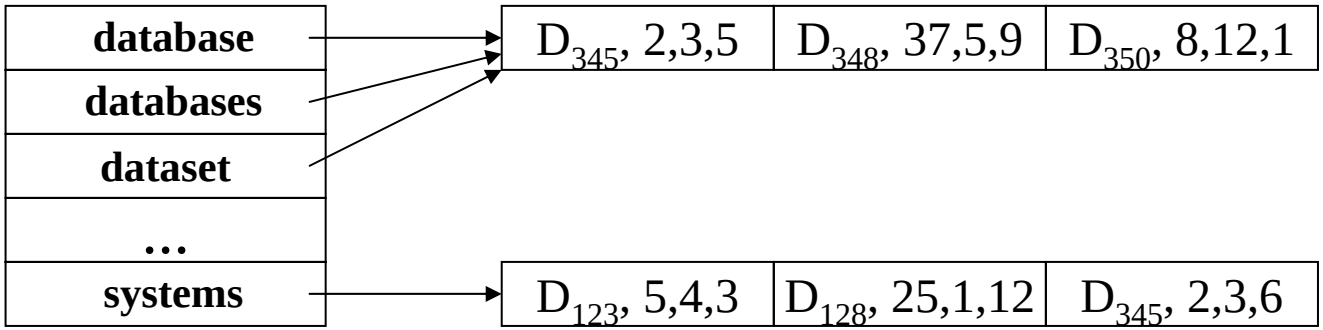
- 可保存出现频率，以便支持基于统计的检索:

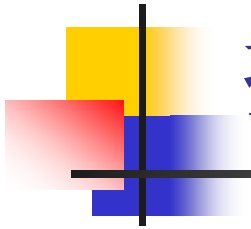


- Postings中的第二个单元可以是该term的权重 (例如, 可以被归一化在0和1之间)，或者是该term的出现频率

扩展 —— 同义词

- 同义词对于提高召回率很有意义
- 同义词可以通过指针指向同一个postings list.



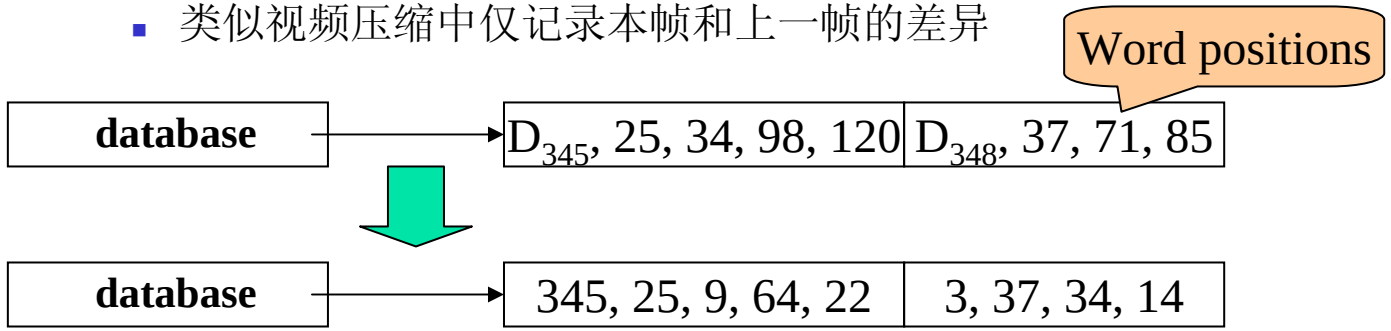


扩展 – Term截断

- 后缀截断是stemming的简单形式:
  - comput*
    - computer, computing, computation, etc.
  - 如果倒排索引用trie来实现，则很容易进行后缀截断
  - 在b+树上进行这种操作就有点难度 (例如：可能需要使用一个映射表将comput*映射为compute, computer, computing等)
  - 如果使用hash，则不会太灵活

倒排文件中的研究问题

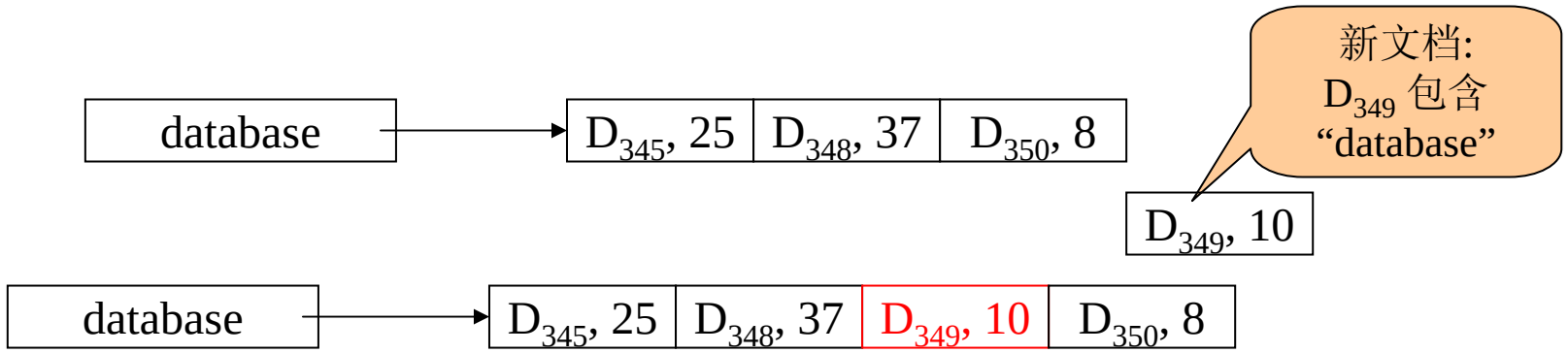
- 索引和postings的压缩问题
  - 分配给文档ID和词位置的字节数
  - 16 bits不够，32 bits又浪费空间
  - 仅记录相邻文档ID和词位置之间的差异
    - 类似视频压缩中仅记录本帧和上一帧的差异

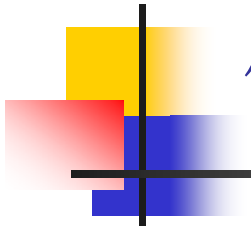


- 快速插入的方法
  - 批更新：为新的更新和插入生成一个空索引，然后把它融合到主索引表中

倒排索引的开销

- 一个文档插入时最坏的情况:
  - 当文档包含n个词，并且每个词都不重复，插入时需要更新n个posting表
- 对于posting表中的每个更新操作:
  - 如果postings表没有排序，新的posting项可以被追加到表的末端，更新操作很快，但是检索无序的posting表很慢
  - 如果posting表示排序了的，那么插入一个新的posting项需要很大的开销





倒排索引的负载

- 插入的开销依赖于更新的频度，对索引的即时性的要求，以及系统的工作负荷能力
- 对于图书馆应用来说，插入操作不是问题，对新闻和互联网方面的应用而言，就是一个问题
- 当我们设计一个新的插入操作时，我们必须考虑:
  - 主存和临时存储器中可利用的空间大小
  - 索引的批量和在线更新

删除和更新

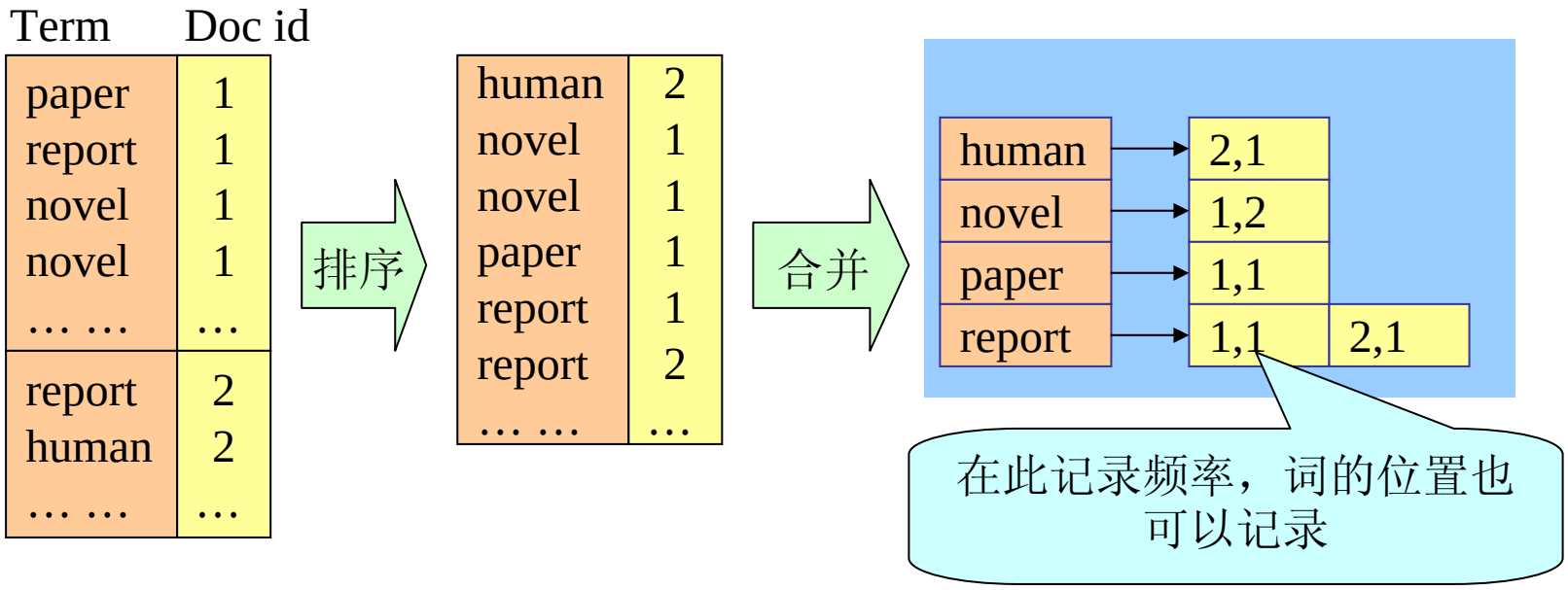
- 更新就是一个删除操作，后面跟着一个插入操作
- 为了支持删除操作，需要维护一个前向索引(forward index)来记录文档中包含的词

- 找到将要被删除的文档ID
- 从前向索引中获得该文档中包含的词
- 根据该文档中的词定位倒排索引中的posting项，并在这些posting项中将该文档ID删除

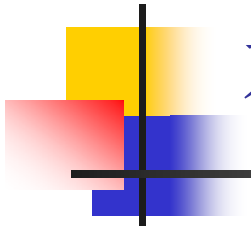
- 删除开销很大; 为了降低删除开销，可以：
  - 维护一个表，表中存放要删除的文档ID号（先不在倒排表上实施）
  - 在检索过程中，忽略那些被登记在表中的文档ID
  - 周期性地对倒排文件进行更新

通过排序进行批插入

- 收集所有将被插入索引中的新文档
- 从每个文档中提取term，并准备一个批倒排文件:



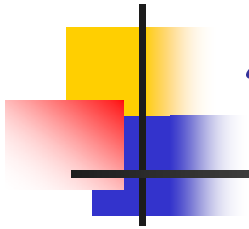
批倒排文件被插入到主倒排文件中 – 批插入为什么会快呢？因为一个词可能出现在多个文档中！



在批插入中提高速度

- 从倒排文件中检索出一个postings表，插入若干posting项，该表再放回倒排文件中
- 批倒排文件相当小，因此创建它并不难
- 批不能太小，否则很少会出现多个文件包含一个词的情况
- 批不能太大，否则批的生成本身开销也很大
- 采用批更新将使对新文档的检索被延迟

到目前为止，我们假定postings lists被存放在单一的顺序文件中，从更新的角度看，这可能不是一个好的方法



快速倒排算法

- 大量文档的倒排操作
- 每批多大合适？
  - 批的大小受到主存的限制，因为需要对倒排文件进行排序
- 怎样降低倒排文件的随机扩展呢？
  - 给倒排文件与分配磁盘空间

快速倒排算法

Doc id	Word IDs
1	3, 5, 12, 14
2	1, 3, 4, 11, 12
3	2, 4, 5, 12, 13
4	1, 5, 11, 12, 14
5	3, 7, 13, 14

分裂

以前的方法需要对23个项进行排序

分裂为三个相同大小的文件

	load files		
	1	2	3
no. of unique word IDs	4	3	3
no. of word-doc pairs	8	6	9

加载 file 1

1	3
2	1, 3, 4
3	2, 4
4	1
5	3

加载 file 2

1	5
2	11
3	5
4	5, 11
5	7

加载 file 3

1	12, 14
2	12
3	12, 13
4	12, 14
5	13, 14

插入

插入

插入

1	2, 4
2	3
3	1, 2, 5
4	2, 3
5	1, 3, 4
7	5
11	2, 4
12	1, 2, 3, 4
13	3, 5
14	1, 4, 5

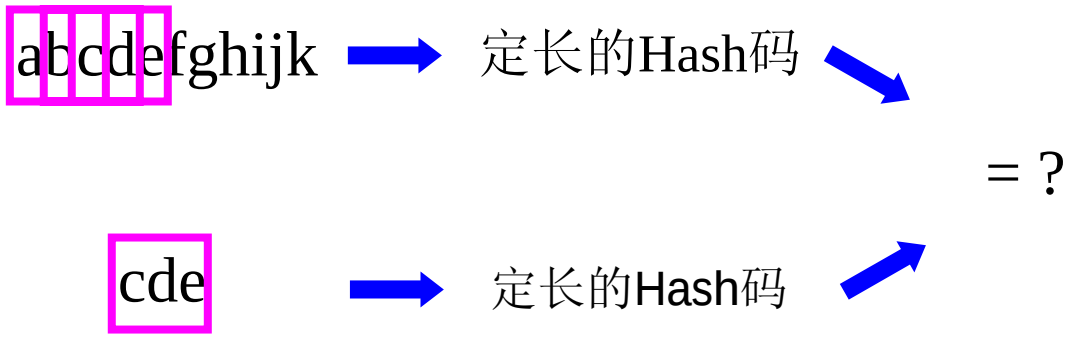
最终的倒排文件

- 词ID总是被追加到后边
- 每一部分有已知的大小，可以预先分配存储空间

索引与检索
——Signature文件

使用Signatures

- 用Karp-Rabin方法进行模式匹配?



- 想法：比较字符串的压缩表示更有效
- 这种表示成为*signatures*

词的Signatures

- 一个词的Signature是一个位向量
  - 由F位组成，其中m位置1

文本串: free text information retrieval

Words	hash(word signatures)
free	001 000 110 010
text	000 010 101 001
information	000 000 011 110
retrieval	101 000 100 001

$F = 12$

$m = 4$

词Signature的生成

- 词signature

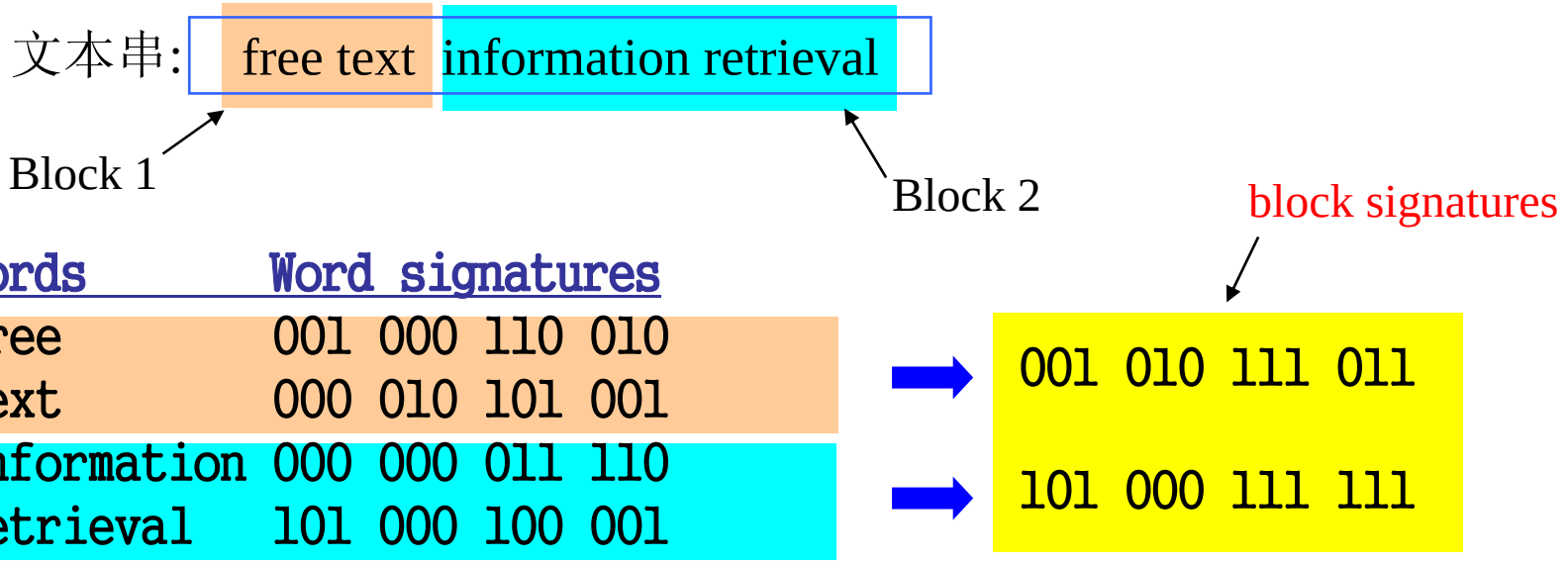
Word	word signature
free	001 000 110 010

将 'free' 转换为ASCII值，然后转换为32位整数i（每个字符8位）
例如：free = 66726565 (hex)

```
F位全置0
srandom(i)                          初始化随机种子
j=0
while j < m
do p = random();                    生成32位随机数
   p = p mod F                      映射到0和F-1之间
   if (signature(p) == 0) {         确保m位置1
      signature(p) = 1
      j++
   }
done
```

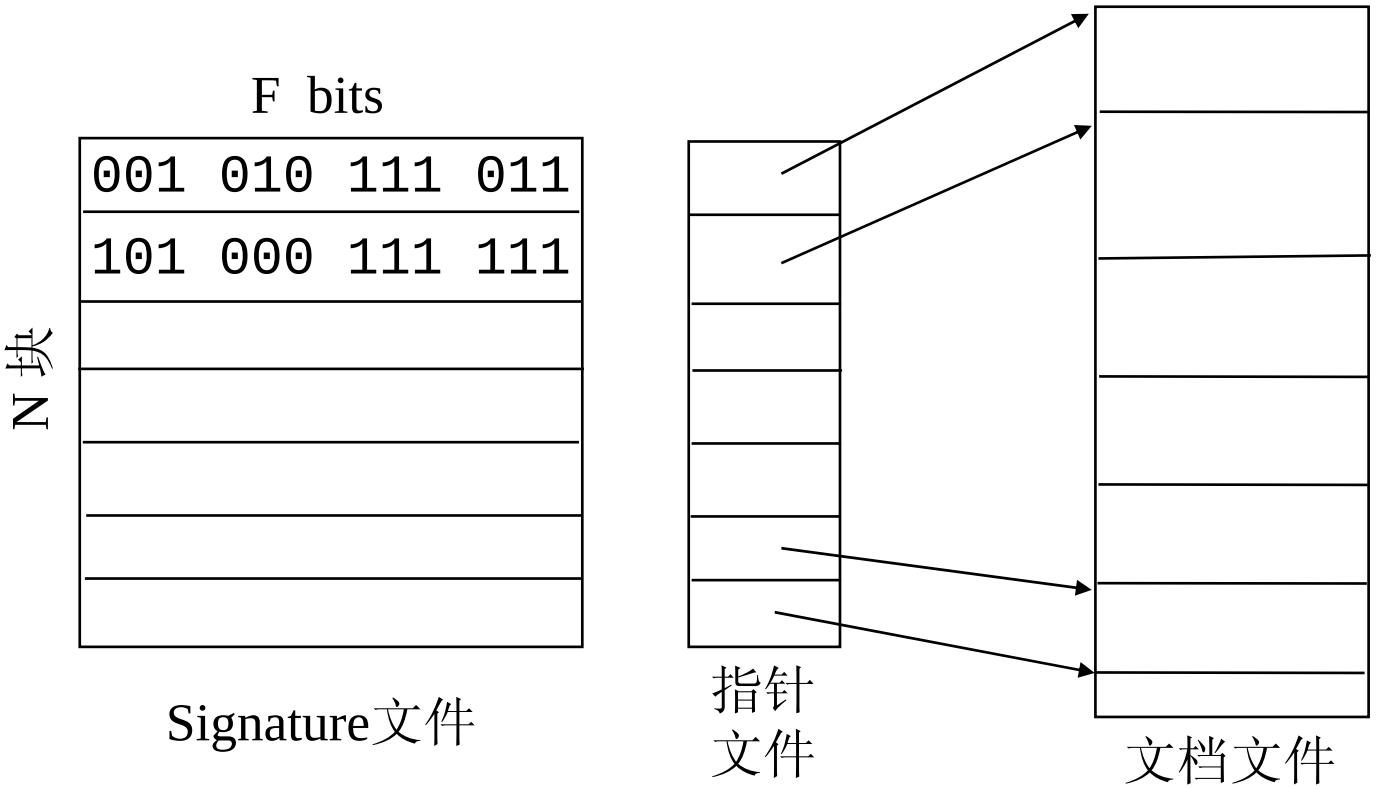
块Signature生成

- 一个文档 (长串)被划分为固定大小的块
  - 假设一个块包含两个连续的词
- 重叠(superimposed)编码
  - 块中所有词的signature按位进行“或”操作



顺序的Signature文件

- 最直接的方法就是按顺序存放Signature文件



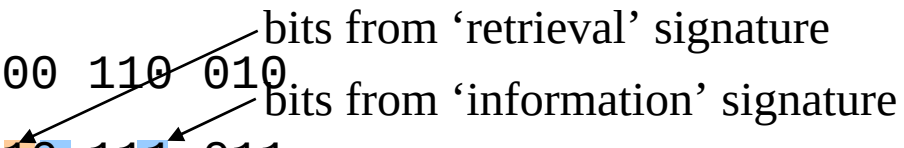
用Signature文件进行检索

- 给定一个*q*, 产生一个查询式的signature (*F* bits with *m* 1's) *Sq*
- 匹配条件：*Sq* ∩ *Sb* = *Sq* ，例如： 如果一个块在q的signature取1的位置上也取1，则该块被返回
  - 例如：
    - query signature q 000 010 101 001
    - block 1 001 010 111 011
    - block 2 101 000 111 111

Block 1: 000 010 101 001 ∩ 001 010 111 011 = 000 010 101 001
Block 2: 000 010 101 001 ∩ 101 100 111 111 ≠ 000 010 101 001
块1匹配成功，块2匹配失败

False Drop概率

- 但是… 如果一个query signature和一个 block signature匹配成功, 就一定能够确保是一次正确的匹配吗?
- 假设query为 *free*
  - `query signature  001 000 110 010`
  - `block 1  001 010 111 011`
  - `block 2  101 000 111 111`
  - block 1和 block 2均匹配成功，被返回，然而在block 2中并未出现*free*



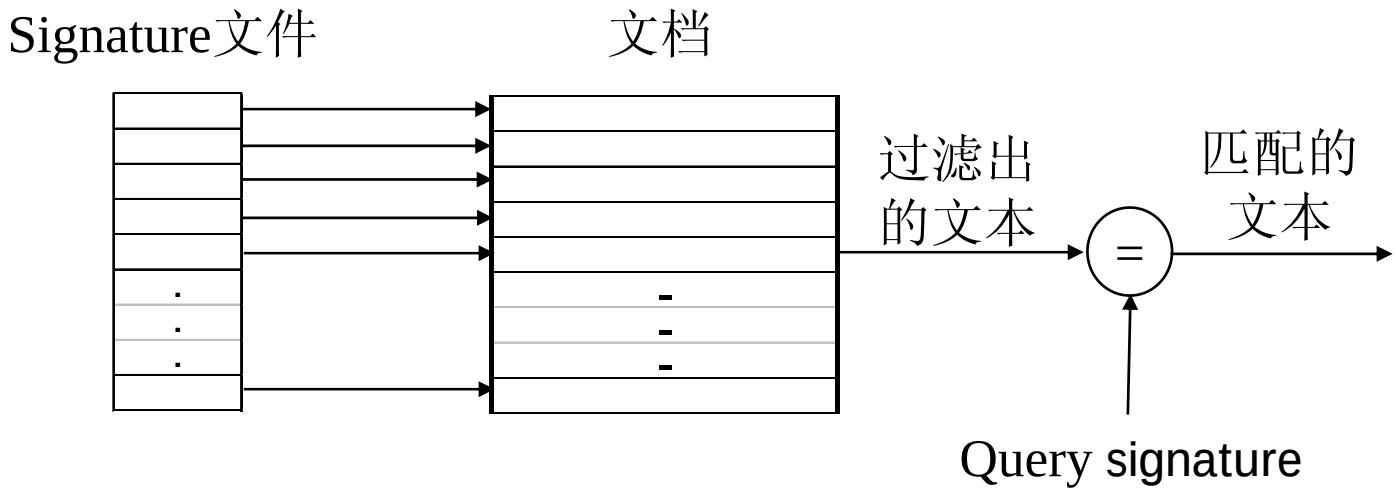
- 此时，Block 2 称为false drop.
- What causes false drops?
  - 主要原因：不同词signature的重叠
  - 次要原因：hash冲突(两个不同的词具有相同的signature), 如果*F*足够大，hash冲突的可能性很低

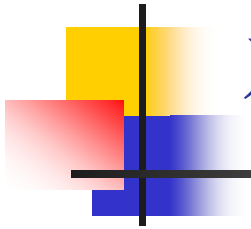
False Drop概率

- ***False drop** 概率*: 一个文本块根据其signature看包含某个term，但事实上并不包含的概率
  - F_d 依赖于signature中1的个数
  - 问题: signature中1越多越好，还是越少越好?
- 太多位置1
  - False drop概率上升，极端情况，所有位都是1，将匹配任何query
- 置1的位太少, 过滤能力下降
  - 没有充分利用signature提供的空间
- 最优的情况是1和0的数量相等
  - 直觉的解释：当n/2被置1后，可能匹配的情况最多
  - 例如：假设signature是4位，则其中2位置1时可能匹配的模式是6个，最多
  - $C_4^1=4$, $C_4^2=6$, $C_4^3=4$

作为过滤器的Signature文件

- Signature文件可以剔出那些不包含query terms的文档
- 可以将通过signature过滤器的文本和查询式直接比对，从而避免False Drops





存储开销

- 存储开销决定于被散列到一个signature中的词数

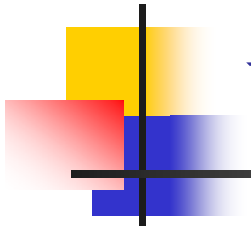
$$M = nF$$

M: 需要的存储位;

n: block数

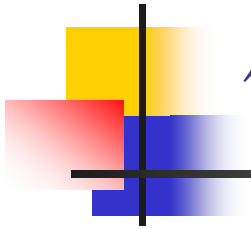
F: 每个signature的位数

- 关键问题: 对于一个包含了给定词数的文档，生成多少signature比较合适?
- 如果更多的词散列到一个signature中，则存储的开销将降低，而false drop的概率将升高



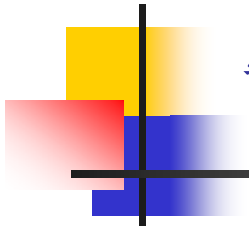
设计时的决策

- 对signature file的需求
  - 能够负担的存储开销
  - 能够忍受的false drop概率
- 需要确定的参数:
  - Signature的长度
  - 要将多少个词散列到一个signature中
  - 每个词多少位
- 优化设计:
  - $F_d = 2^{-m}$ $m = F \ln 2/D$
    - F_d - false drop概率
    - m – 置1的位数
    - F – signature的长度
    - D – 一个块中的词数



优点

- Signature文件小而可控
- 由于文件组织简单，因此维护费用小(更新和删除)
- Signatures容易生成，插入费用低
- 重叠编码适合多属性检索
- Signature文件在倒排文件和全文扫描之间做了空间和时间的平衡
  - 适合中等大小的数据库和查询频率较低的系统
- 容易进行并行处理
- 可应用于过滤系统



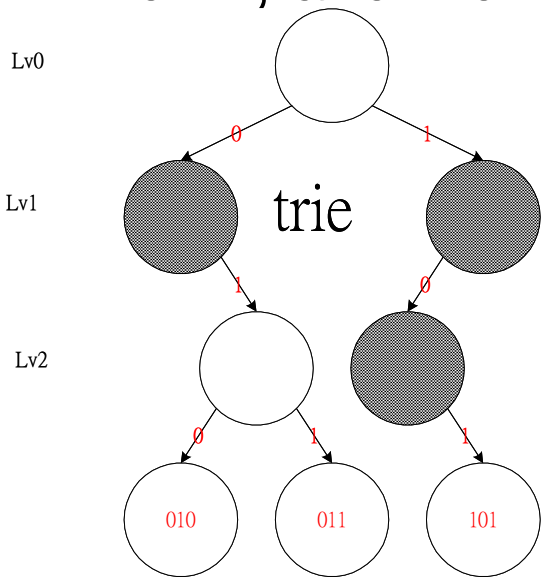
缺点

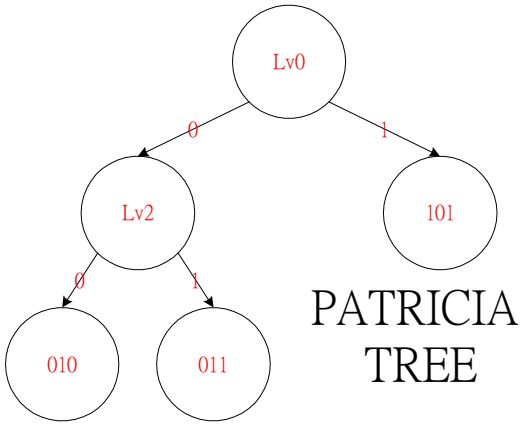
- 和倒排文件相比，搜索速度慢 (尽管比全文扫描快). 对于顺序文件, 所有的signature都必须被比较
- 去除False drops需要昂贵的开销
  - 因为所有被匹配的signature必须通过模式匹配来确认
- 在signature中，很难对频率和权值信息进行编码
- 其它query函数，例如分离条件、同义词、通配符, 邻近(proximity)操作都很难使用

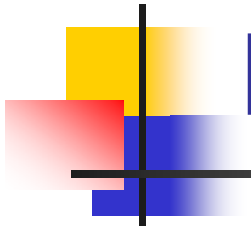
索引与检索
——PAT Tree

PATRICIA TREE

- 存储一个文档的全部“半无限串sistring” (semi-infinite string)
- 一种特殊的 “trie”
- 举例, trie and PATRICIA TREE with content ‘010’, ‘011’, and ‘101’.

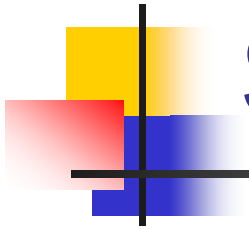






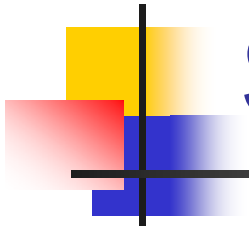
PATRICIA TREE

- 因此PATRICIA TREE将在它的内部节点中包含一下属性:
  - 索引位 (检查位)
  - 子指针 (每个节点必须包含恰好2个字节点)
- 另一方面，叶节点必须存储实际用于比较的内容



SISTRING

- 也可以说是“后缀串”
- 二值位模式
- 在理论上无限长
  - 110010000…
  - 10010000…
  - 0010000…
  - 010000…
  - 10000…
- 实际上我们无法无限地存储它，对于上例，我们只需存储5位长，它们已经足够将这些串区别开来

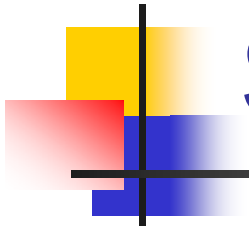


SISTRING

- “位”级太抽象，依赖于应用，我们很少应用到“位”级，字符集是一个比较理想的想法
- 例如：CUHK
  - 相应的sistrings为：
    - CUHK000…
    - UHK000…
    - HK000…
    - K000…
  - 我们需要每个至少为4个字符

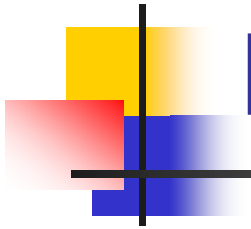
子串的存储开销

- *n* 长字符串有*n(n+1)/2* 个子串，最长子串长度为n
  - 例如：‘CUHK’ 是4个字节长, 由4*5/2=10 个不同的子串构成: C, U, …, CU, UK, …, CUH, UHK, CUHK.
  - 全部子串的存储开销为O(n^2)max(length)，即 O(n^3)



SISTRING 的存储开销

- SISTRINGs 在存储子串方面很有效
- 用SISTRING存储 ‘CUHK’需要O(n^2)的开销，因为SISTRING有n个
  - CUHK <-同时表示了 C CU CUH CUHK
  - UHK0 <-同时表示了U UH UHK
  - HK00 <-同时表示了H HK
  - K000 <-仅表示K
- 对sistrings的前缀匹配等价于对子串sub-strings进行搜索
- 结论：sistrings表示方法比 sub-string优越

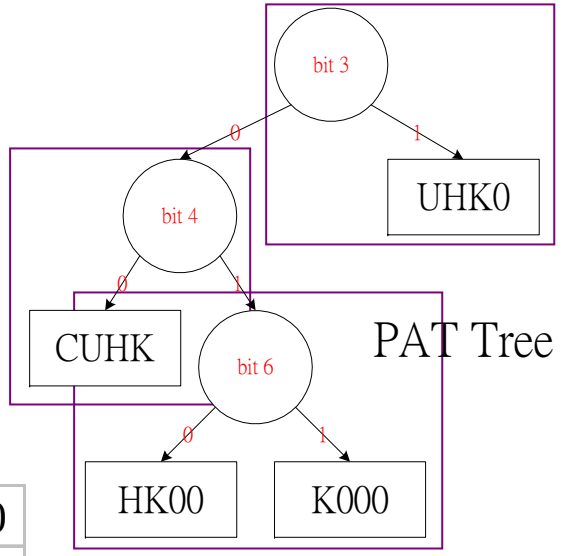


PAT Tree

- 如果一个文本只包含 ‘CUHK’，情况如何?
  - 我们希望在字符级上进行操作，但是PAT运行在“位”级上。因此，我们必须知道每个子串的位模板，以便了解PAT树的实际情况
  - 在“位”级进行可以保证PAT树上的每个节点都有两个儿子0和1，从而构成二元树，便于操作

PAT Tree (举例)

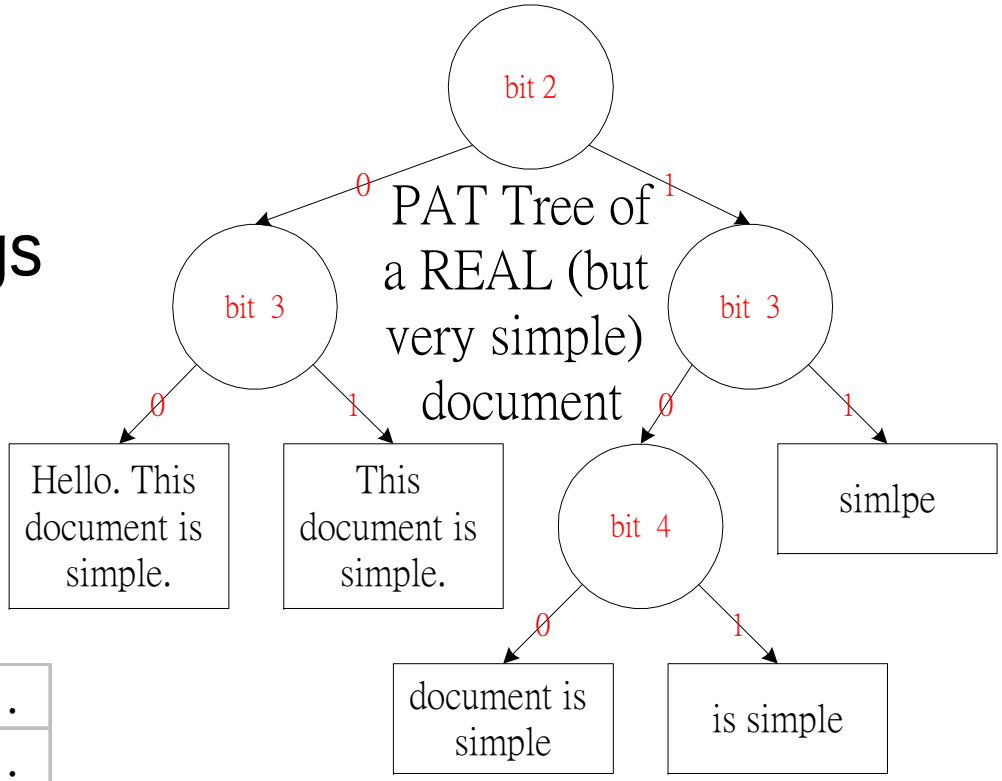
- 下面是四个sistring的实际位模板



CUHK	01000011 01010101 00000000 00000000
UHK0	01010101 01001000 00000000 00000000
HK00	01001000 01001011 00000000 00000000
K000	01001011 00000000 00000000 00000000

PAT Tree (举例)

- 需要$O(n_2)$存储sistrings

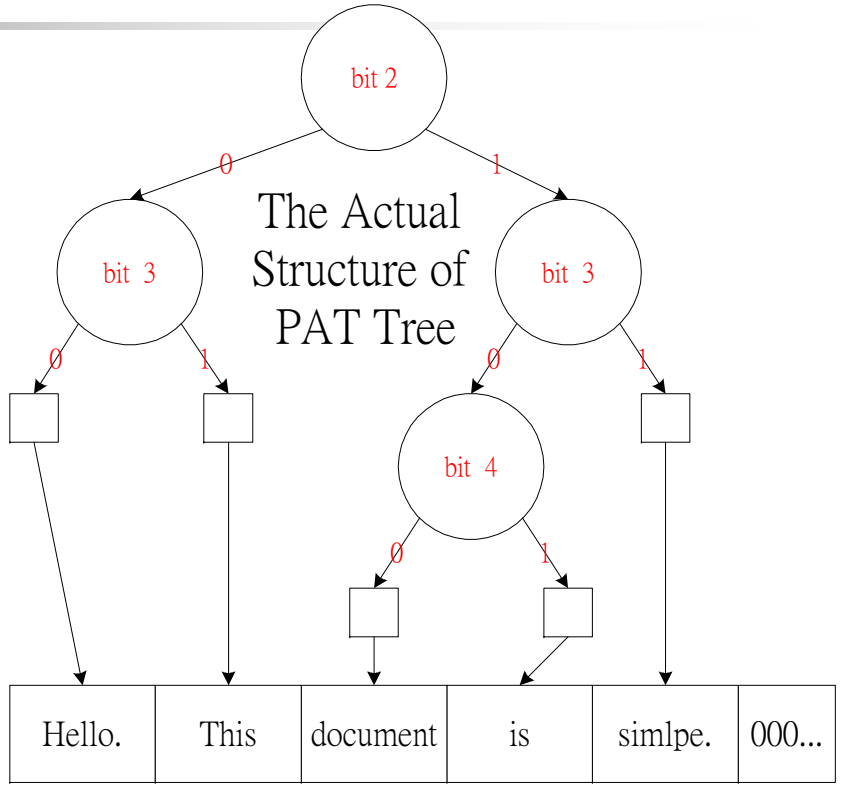


Hello This document is simple	01001000 …
This document is simple	01010100 …
document is simple	01100100 …
is simple	01101001 …
simple	01110011 …

此时把文档看成词串！

PAT Tree (实际的结构)

- 我们只需维护文档本身
- PAT Tree象一个索引结构
- 存储开销
  - 文档 O(*n*)
  - PAT Tree索引O(*n*)
  - 叶节点指针O(*n*)
- 因此，PAT Tree是一个线性数据结构！



第1步：SISTRINGS

.can .a.can.can.cans?

STRING: . c a n . a . c a n . c a n . c a n s ?

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

SISTRING	OFFSET
.can.a.can.can.cans?	0
can.a.can.can.cans?	1
an.a.can.can.cans?	2
n.a.can.can.cans?	3
.a.can.can.cans?	4
a.can.can.cans?	5
.can.can.cans?	6
can.can.cans?	7
an.can.cans?	8
n.can.cans?	9
.can.cans?	10
can.cans?	11
an.cans?	12
n.cans?	13
.cans?	14
cans?	15
ans?	16
ns?	17
s?	18
?	19

第2步：排序并找出最小的有区别力的前缀

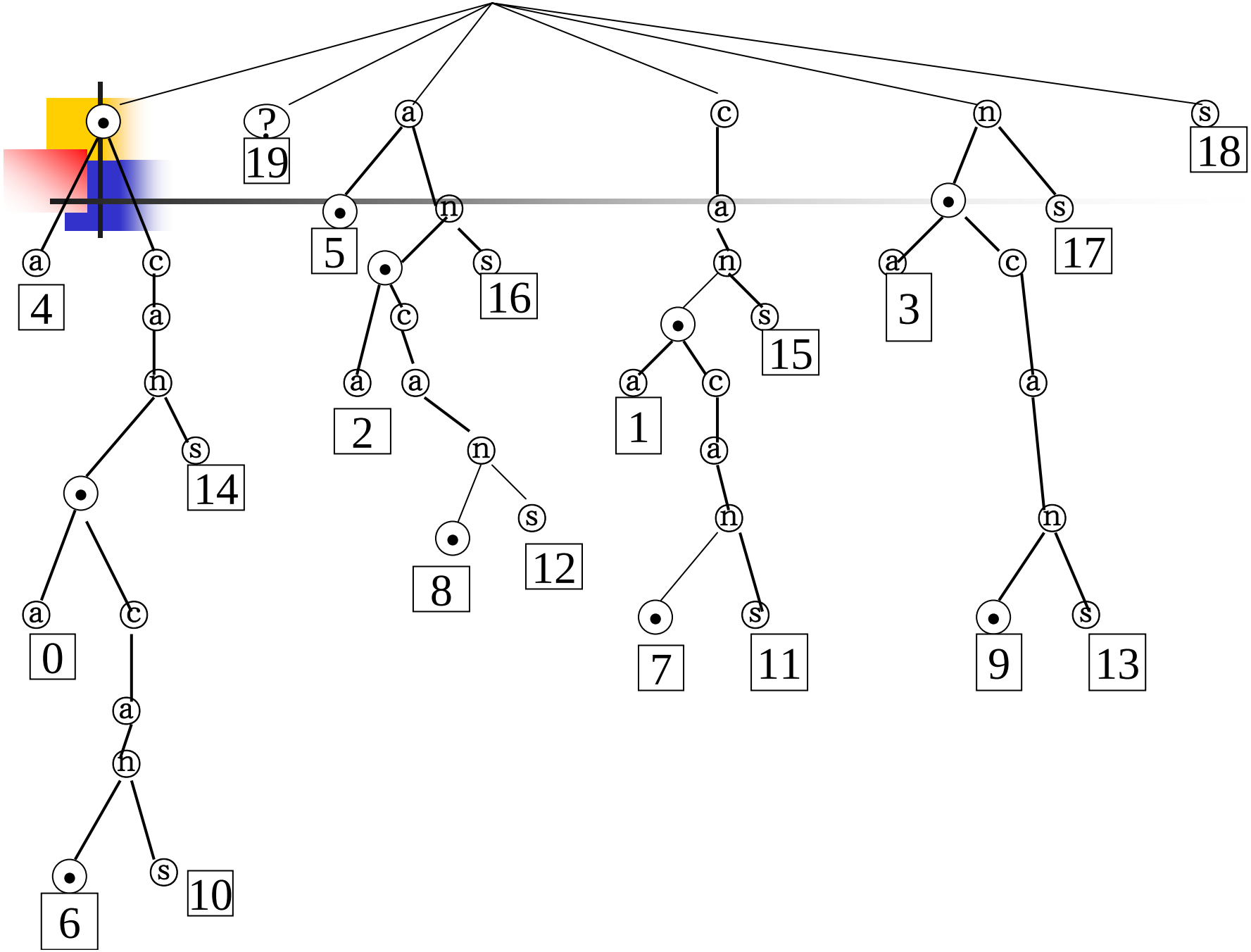
.can .a.can.can.cans?

STRING : . c a n . a . c a n . c a n . c a n s ?

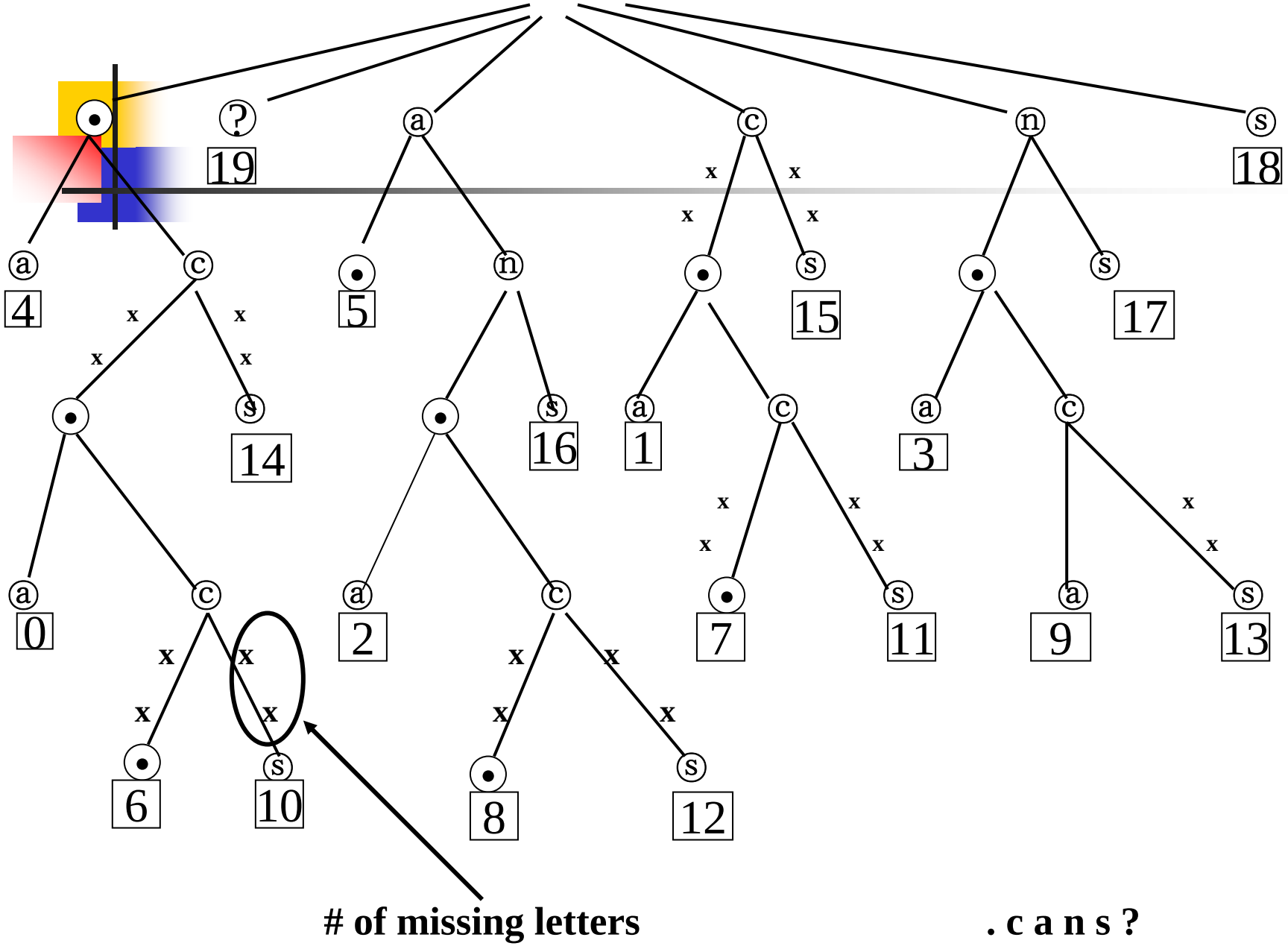
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

SISTRING	OFFSET	MINIMAL DISTINGUISHING PREFIX
.a.can.can.cans?	4	.a
.can.a.can.can.cans?	0	.can.a
.can.can.cans?	6	.can.can.
.can.cans?	10	.can.cans
.cans?	14	.cans
?	19	?
a.can.can.cans?	5	a.
an.a.can.can.cans?	2	an.a
an.can.cans?	8	an.can.
an.cans?	12	an.cans
ans?	16	ans
can.a.can.can.cans?	1	can.a
can.can.cans?	7	can.can.
can.cans?	11	can.cans
cans?	15	cans
n.a.can.can.cans?	3	n.a
n.can.cans?	9	n.can.
n.cans?	13	n.cans
ns?	17	ns
s?	18	s

第3步：从前缀中产生Trie

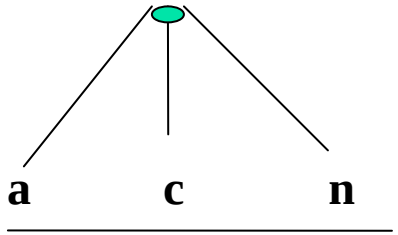


第4步： 通过使用skipped bits简化树

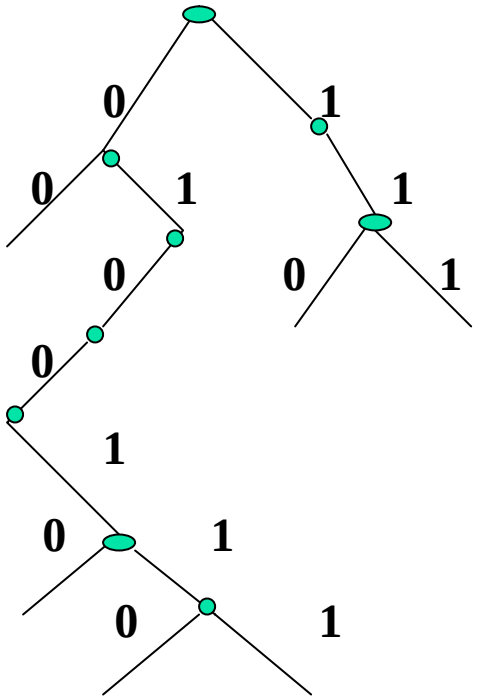


将字符转换为ASCII码，

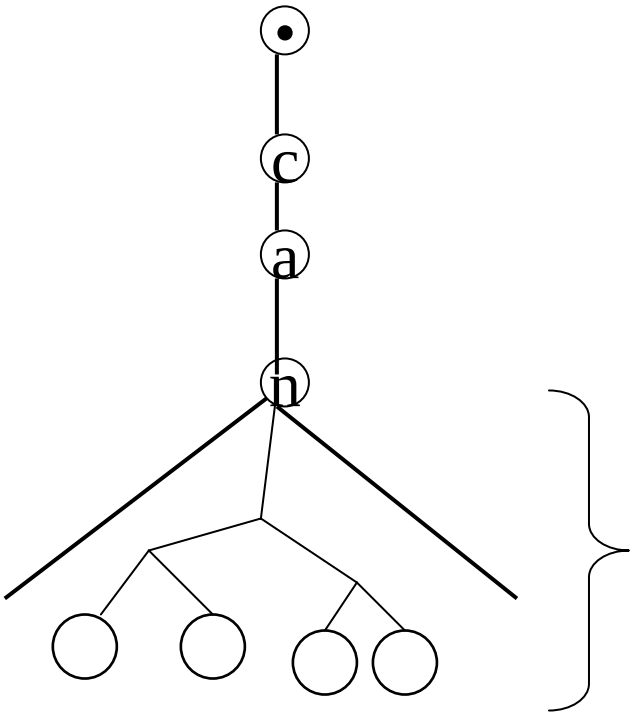
将基于字符的树转换为0-1数字化的树



a	c	n
0	0	0
0	0	0
0	0	0
1	1	1
0	0	1
1	1	0



前缀搜索

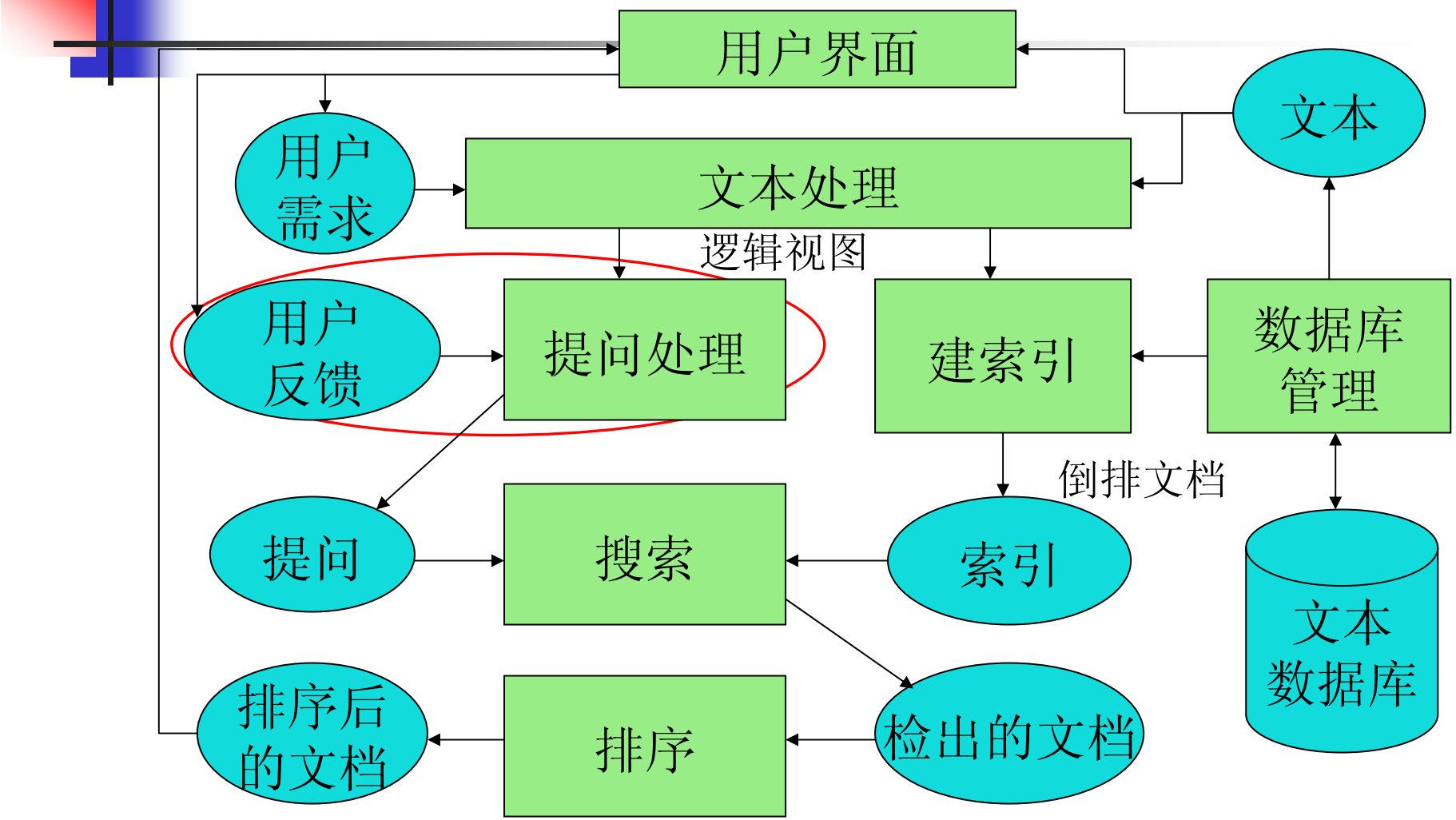


如果没有分支和query中的下一个字符匹配，则失败.

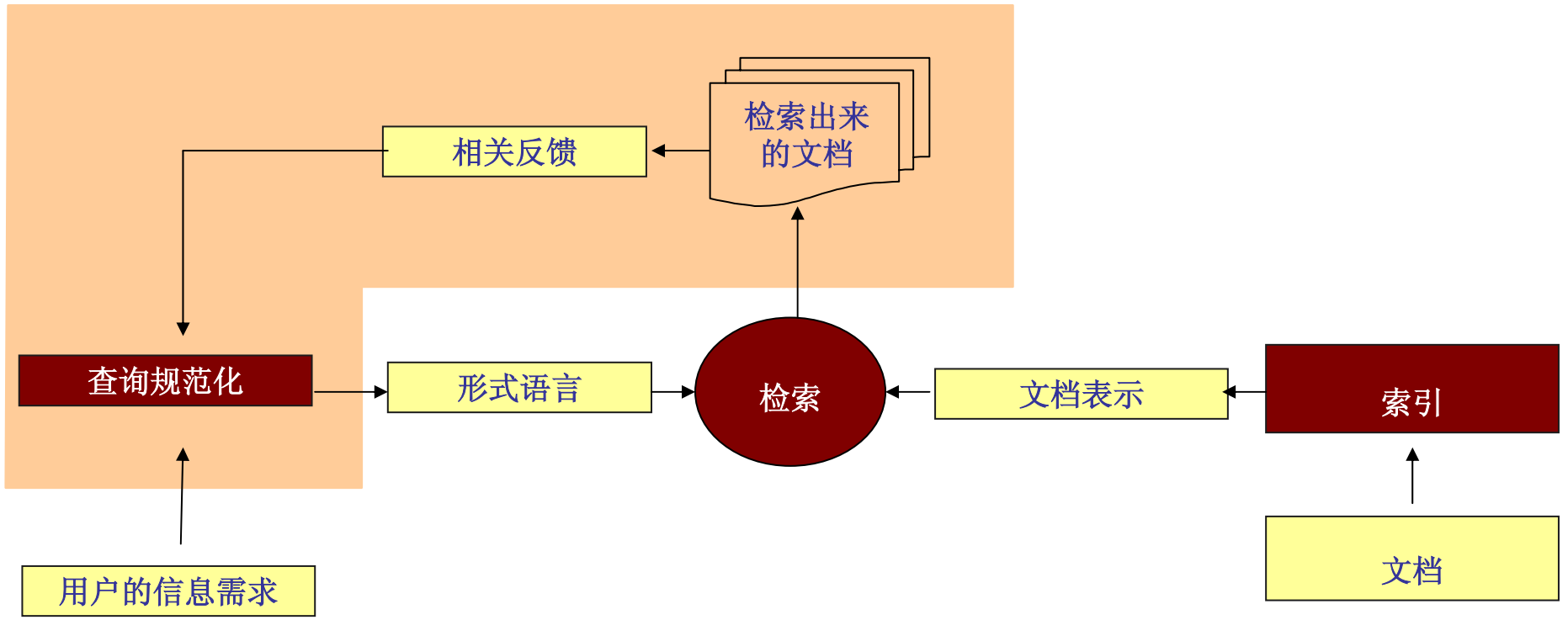
列举出饱含该前缀的所有串

Query处理
——相关反馈

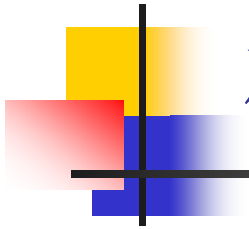
信息检索系统的体系结构



文档检索

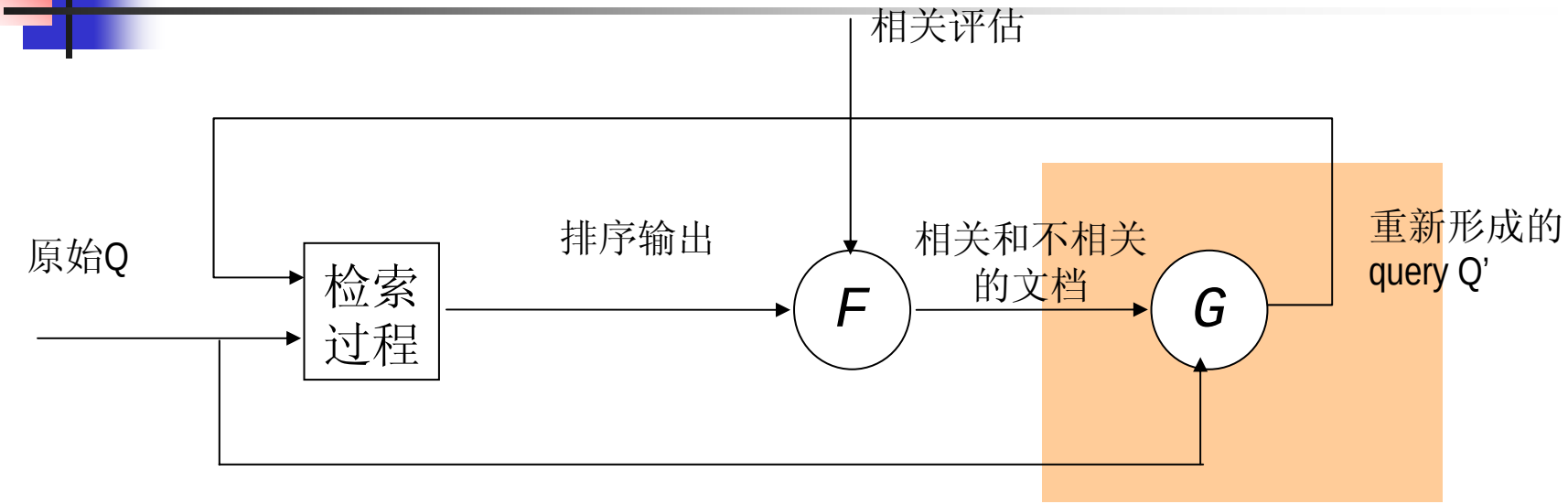


相关反馈Relevance Feedback



- 手工的query再生成很难控制
  - 相关文档和不相关文档的特征不明显
  - 文档特征很难被转化为正确的query形式
- *相关反馈* – 根据用户对文档的相关性评估产生新的查询

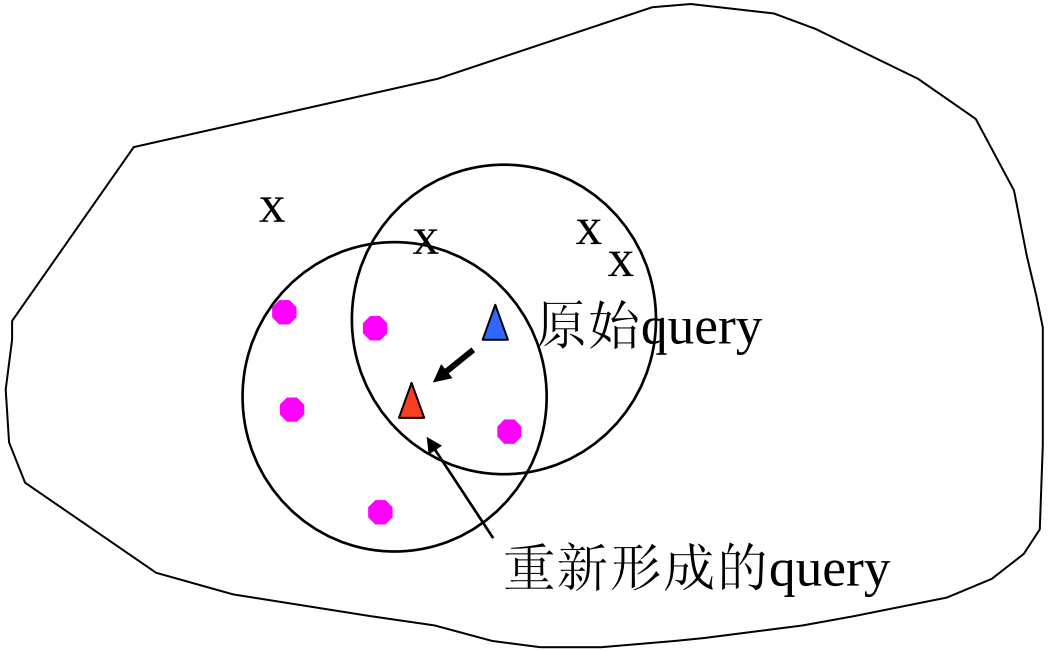
Query修改过程



F: 从用户那里接受相关性评估，输出相关文档和不相关文档

G: 实现相关反馈公式

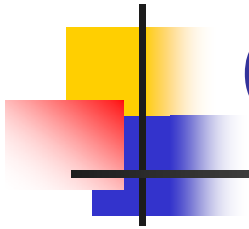
相关反馈的作用



● 相关文档

x 不相关文档

根据原始的query
检索出5篇文档

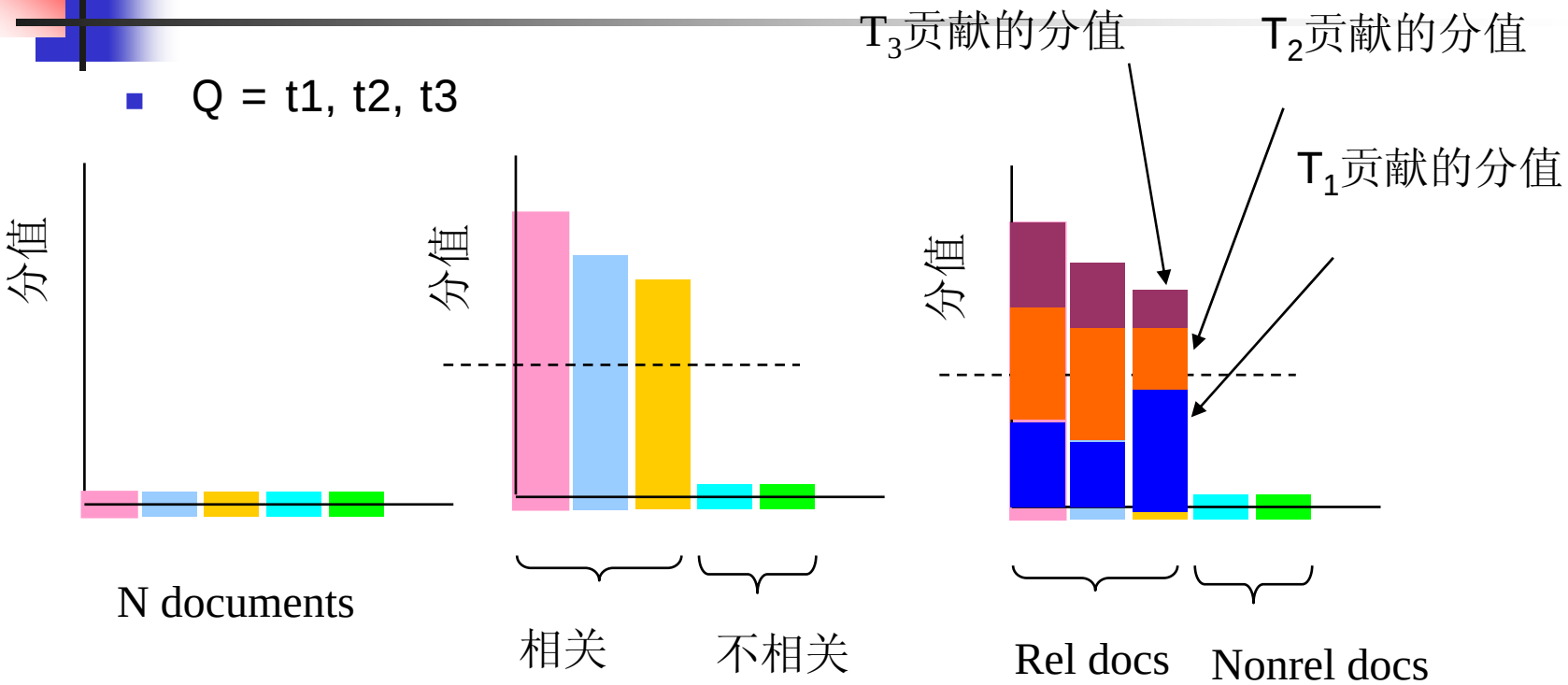


Query修改的基本思路

- 出现在相关文档中的terms被添加到原始的query向量中，或者这些term的权重在创建新的query时有某种程度的增长
- 出现在不相关文档中的terms被从原始query中删除，或者这些term的权重某种程度地降低

理想情况

- Q = t1, t2, t3



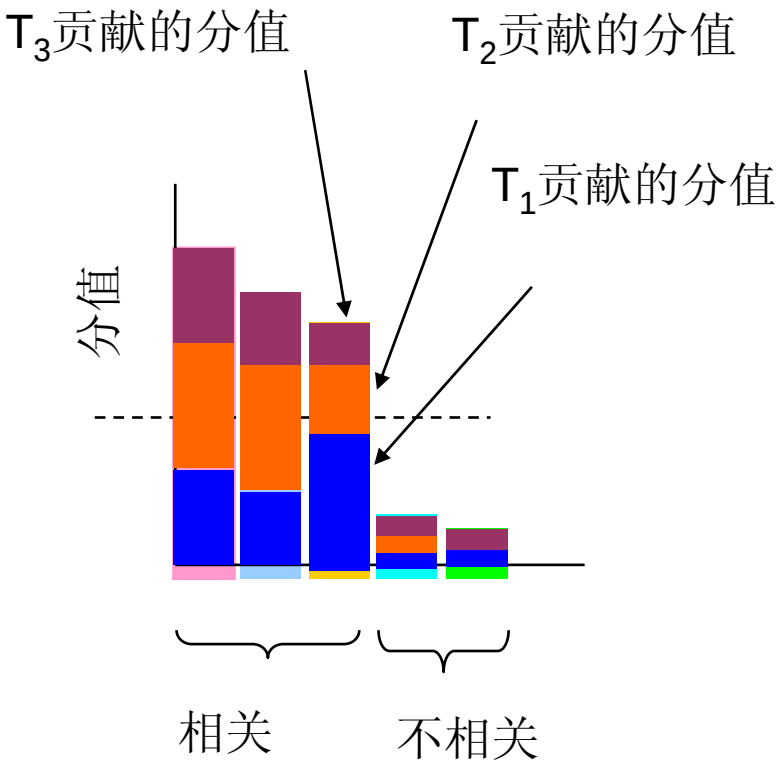
initial

处理query term后

理想情况：query terms只出现在相关文档中!

一般情况

- Q = t1, t2, t3



一般来说，一个term可能在相关文档和不相关文档中都出现。问题是：“是否应该应该在query中包含它，如果包含，怎样打分”

优化的Query

- 根据已知的相关文档集*DR*和不相关文档集*DN*，令let t_{ik} 表示term *k* 在文档i中的权重，term *i* 在两个集合中的平均权重分别为：

$$\frac{1}{R}\sum_{i\in D_R} t_{ik} \quad 和 \quad \frac{1}{N}\sum_{i\in D_N} t_{ik}$$

在优化后的query *Qopt*中term *k* 的权值定义为：

$$(Q_{opt})_k = C(\frac{1}{R}\sum_{i\in D_R} t_{ik} - \frac{1}{N}\sum_{i\in D_N} t_{ik})$$

- 问题是：*DR* 和 *DN* 均不知道

考虑不同的情况:
如果t_k 仅出现在相关文档中，它的权值非常高
如果t_k 仅出现在不相关文档中，它的权值就小，甚至为副
如果t_k 在两类文档中都出现，它的权值介于中间

Query修改

以上的query修改公式是建立在已知全部相关文档集和不相关文档记得基础上，但是相关反馈只告诉了你“一些”相关或不相关的文档

- 将用户提示的相关文档集*DR'* 和不相关文档集作为对*DR* 和*DN* 的估计,重复地修改query达到优化的目的
- 从初始query开始

$$Q' = \alpha Q + \beta\left(\frac{1}{R'}\sum_{i \in DR'} D_i\right) - \gamma\left(\frac{1}{N'}\sum_{i \in DN'} D_i\right)$$

Q 是初始的query， α, β和γ 是一个合适的常数

Q, Q',D_i均为加权向量

举例

- Q:初始query
- D1: 相关文档
- D2:不相关文档
- α = 1, β = 1/2, γ = 1/4
- 假设:

	T1	T2	T3	T4	T5
Q =	5,	0,	3,	0,	1)
D1 =	2,	1,	2,	0,	0)
D2 =	1,	0,	0,	0,	2)

$$Q' = Q + \frac{1}{2}\left(\sum_{i \in D_{R'}} D_i\right) - \frac{1}{4}\left(\frac{1}{N'}\sum_{i \in D_{N'}} D_i\right)$$

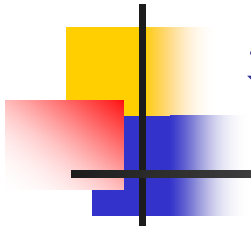
$$S(Q, D_i) = \sum_{j=1}^{t} (Q_j \times D_{ij})$$

$$Q' = (5,0,3,0,1) + \frac{1}{2}(2,1,2,0,0) - \frac{1}{4}(1,0,0,0,2)$$

$$Q' = (5.75, 0.5, 4, 0, 0.5)$$

S(Q,D1) = (5•2)+(0 • 1)+(3 • 2)+(0 • 0)+(1 • 0) = 16
S(Q'D1)=(5.75 • 2)+(0.5 • 1)+(4 • 2)+(0 • 0)+(0.5 • 0)=20

S(Q,D2) = (5•1)+(0 • 0)+(3 • 0)+(0 • 0)+(1 • 2) = 7
S(Q'D2)=(5.75 • 1)+(0.5 • 0)+(4 • 0)+(0 • 0)+(0.5 • 2)=6.75

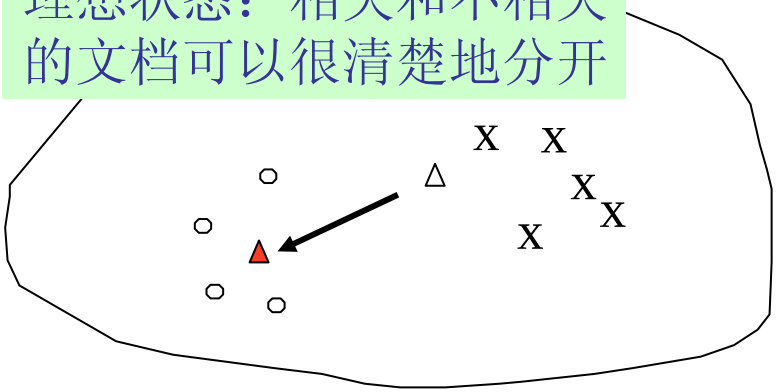


需要确定的参数

- 如何确定α, β和γ？
  - 只考虑相关文档 – 正模型
  - 只考虑不相关文档 – 负模型
  - 同等/不同的权值 – 混合模型
- 相关文档提供的信息比不相关文档更大，因此β应比γ值更大
- 在*R*' and *N*'使用多少文档？
  - 使用全部相关文档和不相关文档
  - 使用全部相关文档和高rank值的不相关文档

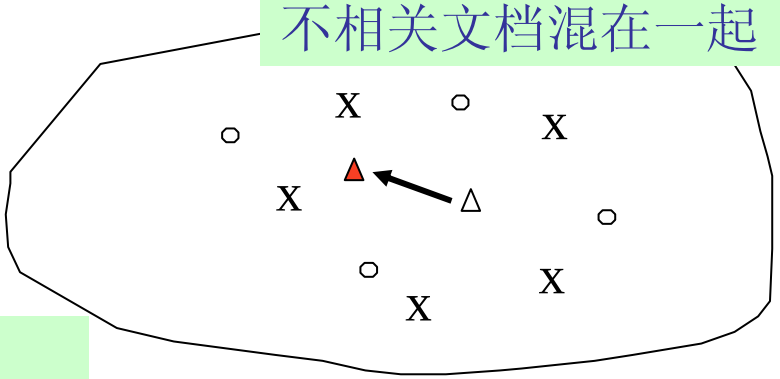
相关反馈的作用

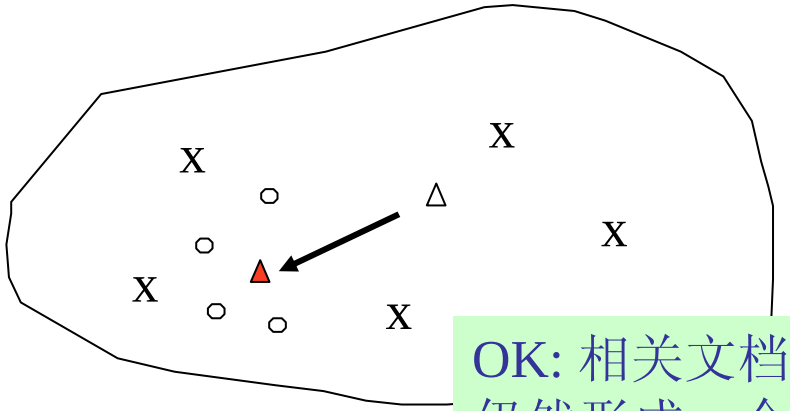
理想状态：相关和不相关的文档可以很清楚地分开



○ 相关文档
x 不相关文档
△ 原始query
▲ 修正后的query

糟糕的情况: 相关文档和不相关文档混在一起





OK: 相关文档
仍然形成一个“簇”

Query 分裂

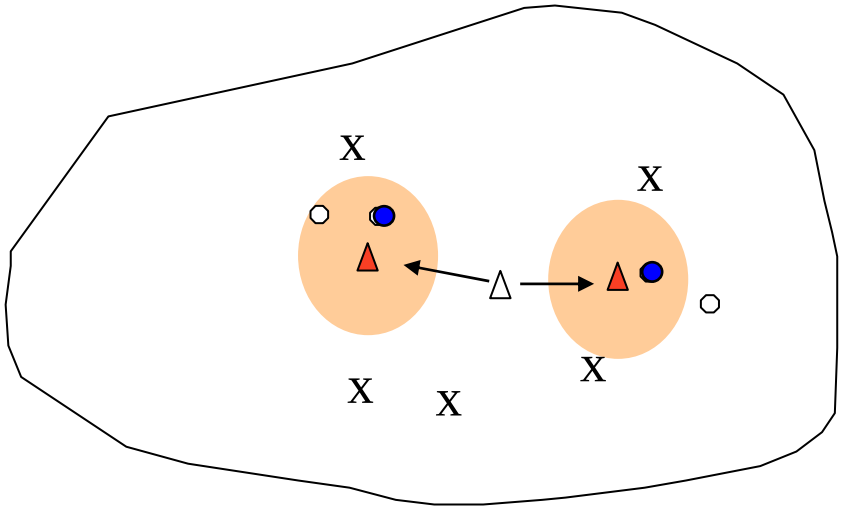
- 当相关文档集R' (或不相关文档集) 没有形成“簇”, 反馈机制无效
  - 当R' 根本不形成“簇”, 我们无能为力
  - 当R' 形成多个“簇”, 如果能发现这些簇, 为每一个簇构成一个新的query，则上面的公式仍然可以使用

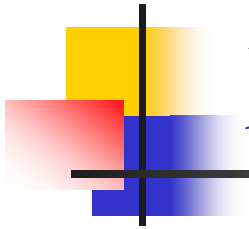
○ 相关文档

x 不相关文档

△ 原始query

▲ 修正后的query

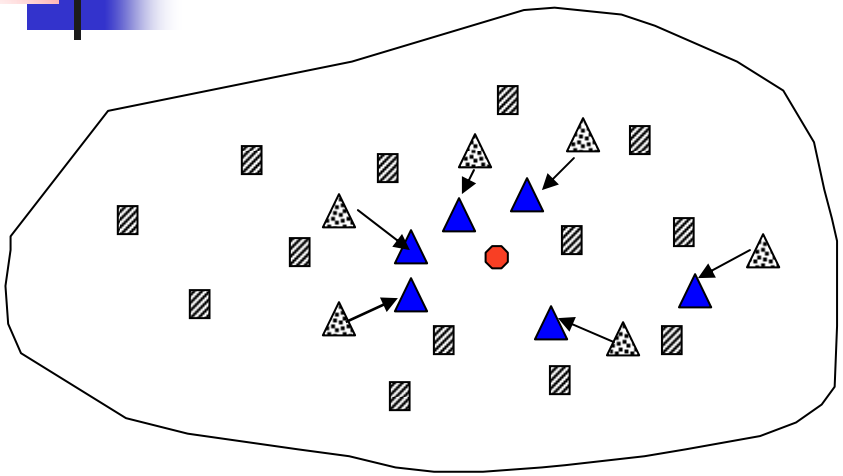




文档空间的修改

- 不修改用户的query, 可以根据用户的反馈修改文档空间 (例如：索引项和文档的权值)
- 思路：使相关文档向量和query的距离拉近，不相关文档向量和query的距离拉远

修改文档空间的例子

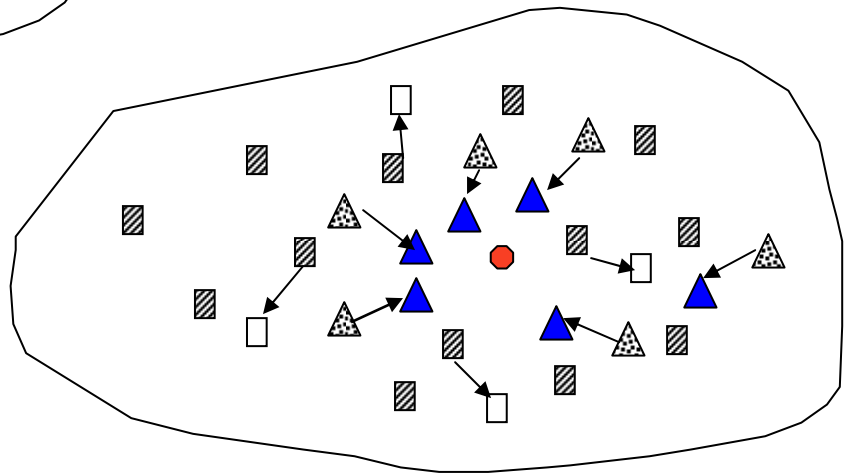


相关文档的修正

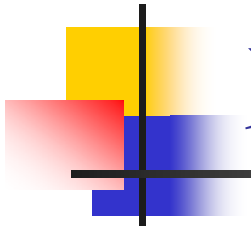
相关文档

不相关文档

query



相关和不相关文档的修改

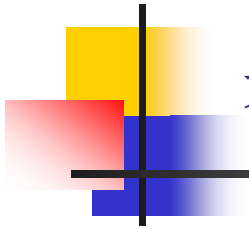


文档的修改

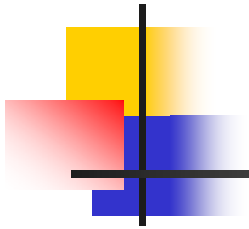
- 对相关文档的修改
  - 被认为是相关的文档在文档空间中一定要和Q更近
    - 将一个不在文档中出现的*query term* 乘以系数α后加入文档中
    - 已在文档中的*query term* 权值乘以系数β，提高权值
    - 不在query中出现的*document term乘以* -γ，降低权值.
  - 使文档向量和query越接近，这些文档向量之间也越接近
- 对不相关文档的修改恰好相反
  - 在query中出现的词权重下降
  - 不在query中出现的词权重上升

Query处理
——查询扩展

基于Thesaurus的Query扩展



- 将query中的term *t*扩展为词典中的同义词synonyms 和相关词related words
- 扩展词的权重weight应低于原始query tems
- 一般将提高召回率recall.
- 可能大幅度降低准确率precision, 特别是对于有歧义的词



同义词词林

- 《同义词词林》
  - 梅家驹、竺一鸣、高蕴琦、殷鸿翔，上海辞书出版社1983 (第一版)，1996年 (第二版)
- 架构
  - 12大类，94中类，1428小类，3925个词群
- 12大类
  - A人，B物，C时间和空间，D抽象事物
  - E特征，F动作，G心理活动，H活动
  - I现象与状态，J关联，K助语，L敬语
- 举例：
  - “苹果” Bh07，“香蕉” Bh07，“西红柿” Bh06，……

Thesaurus的使用：例子

- 五个term的term-term相关矩阵*A, B, C, D*, and *E*.
  - 假设根据阈值将相似度转化为二值的
- 相应的term关联

Original term	Associated terms
A	B
B	A,D
C	E
D	B,E
E	C,D

	A	*B*	*C*	*D*	*E*
A	1	1	0	0	0
B	1	1	0	1	0
C	0	0	1	0	1
D	0	1	0	1	1
E	0	0	1	1	1

使用Thesaurus的例子

- 通过增加相关terms来扩展query，相关term的权值系数为0.5

$$q = \begin{pmatrix} A=4 \\ B=2 \\ C=1 \\ D=1 \\ E=0 \end{pmatrix}$$

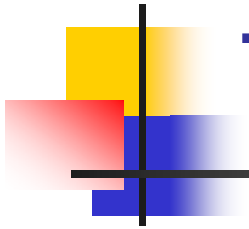
Add B=2
Add A=1,D=1
Add E =0.5
Add B=0.5, E=0.5
Add nothing

Original term	Associated terms
A	B
B	A,D
C	E
D	B,E
E	C,D

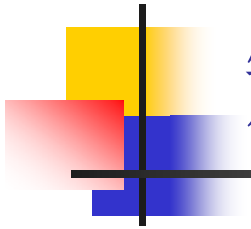
$$q' = \begin{pmatrix} A=5 \\ B=4.5 \\ C=1 \\ D=2 \\ E=1 \end{pmatrix}$$

根据原始query不能找出仅包含E的文档，但新的query可以

Term Phrase Formation

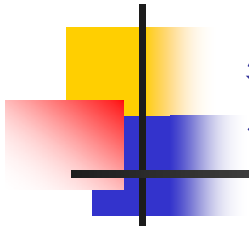


- 单个term作为索引存在一些问题：
  - 单个term的含义要么太特殊，要么太宽泛
  - 单个term没有上下文，容易产生歧义
- 通过增加相关词可以提高召回率，为了同时保证准确率需要使用特殊的term或者将多个term联合使用
- 短语是terms 的结合with particular occurrence properties.



短语头(Head)和短语成员

- 选一个频率很高或者具有negative discrimination value的词作为*phrase head* .
- 短语成员可以是中频或低频词，这些词和短语头有某种同现关系(紧邻或在句子中间隔有限个词)
- 短语不应包含功能词 (stop list)



短语举例

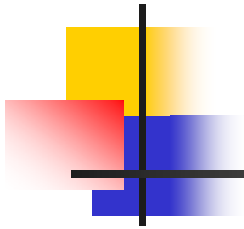
- 文本
  - Effective retrieval systems are essential for people in need of information
- 删除功能词
  - Effective retrieval systems essential people need information
- 选择短语头
  - 假设高频词为: systems, people, information
  - 将它们选为短语头

短语举例 (2)

- 选择短语成员
- 情况1: 头和成员必须紧邻
  - *retrieval systems*, systems essential, essential people, people need, *need information*
  - more restrictive but precise: will miss “effective system”

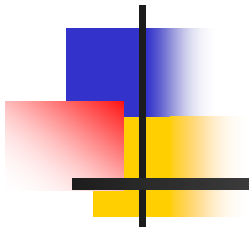
Effective retrieval systems essential people need information

- Case 2: 头和成员在句子中同现即可，不一定紧邻:
  - effective systems, system need, effective people, retrieval people, *effective information*, *retrieval information*, *essential information*
  - 更为积极: 能够产生很好的term，如 “effective system”
  - 对于被语法结构间隔的相关词汇的提取有益，如 “The man driving the BMW is very rich”



通过语言分析构成短语

- 纯粹通过同现提取短语，有可能出错
- 结合语言学分析可能更有效
  - 短语必须和某种句法模式一致
    - 形容词-名词, 名词-名词
      - 例子1*: retrieval systems*, people need, *need information*
  - 短语必须在同一个句子单元中出现
    - 主语短语，宾语短语
      - 例子2: effective systems



谢谢！