Web信息处理与应用

第四节

网页索引

徐童

2024.9.26

上节回顾

- **文档处理的完整流程**

词条化处理

英文：词/词组

中文：分词

停用词处理

规范化处理

词干提取/词形还原

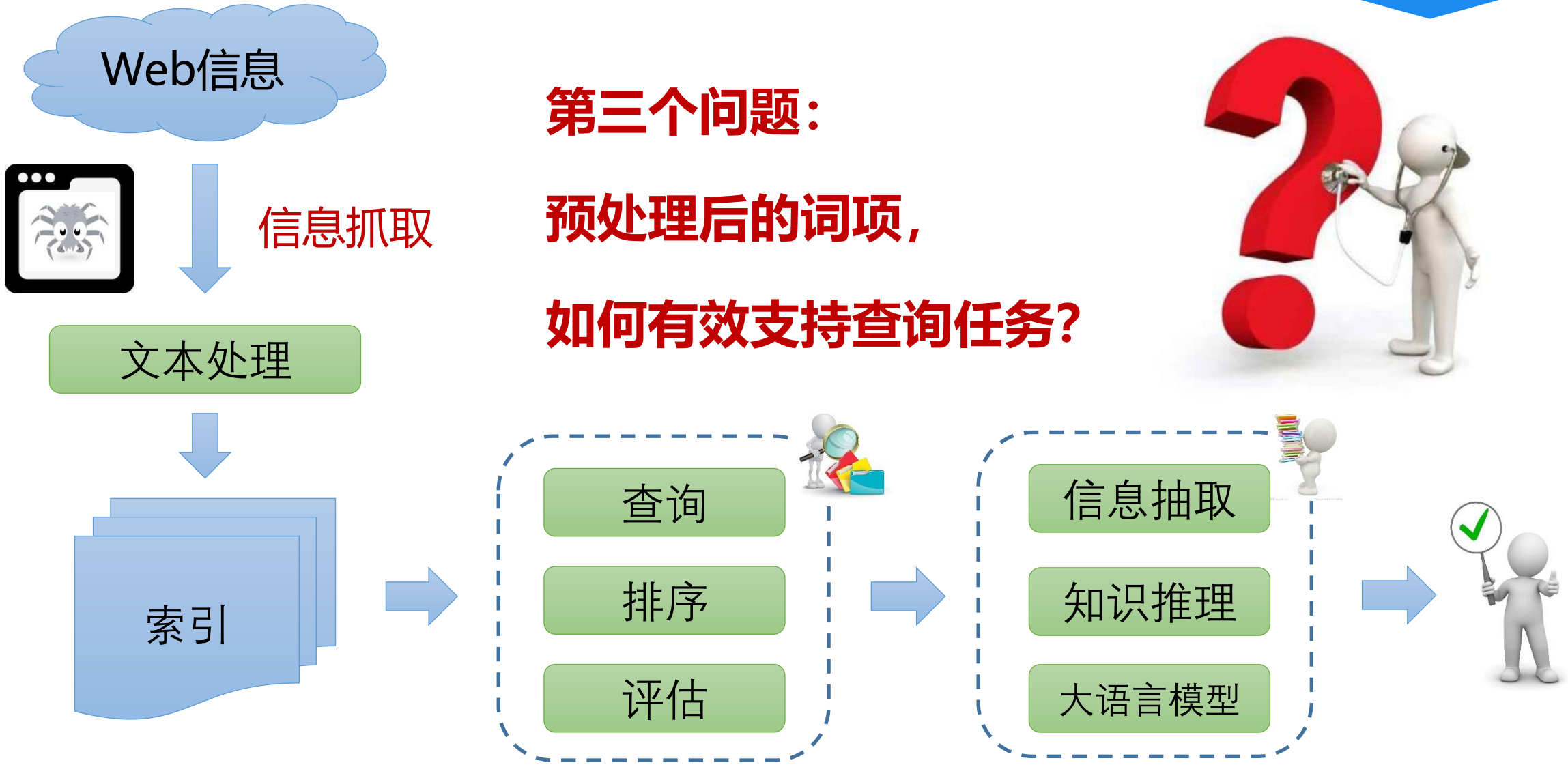
拼写检查、相关词

- **文档处理的完整流程：案例**

- 输入：Friends, Romans, Countrymen, lend me your ears
  - 1）对文档进行分词处理，得到词项
    - Friends / Romans / Countrymen / lend / me / your / ears
  - 2）根据停用词表，删除停用词
    - Friends / Romans / Countrymen / lend / ~~me~~ / ~~your~~ / ears
  - 3）对词项进行规范化处理
    - Friends → **Friend**，Romans → **Roman**，ears → **ear**

本节主题

- 本课程所要解决的问题

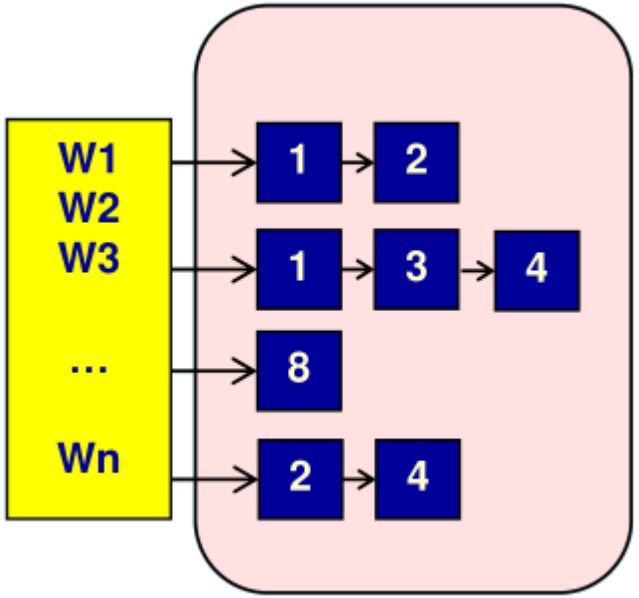


第三个问题：

预处理后的词项，

如何有效支持查询任务？

- **网页索引的目的**



关键词查询 → **索引与匹配** → **返回匹配文档**

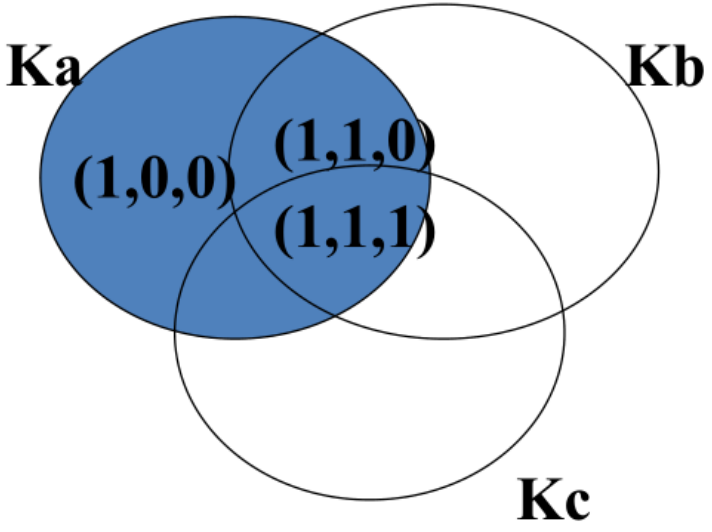
索引的质量，关系着整个搜索引擎系统的精度与效率

本节目录

- **布尔检索**
- 倒排索引
  - 倒排表的构建与查询
  - 倒排表的优化与扩展
- 索引存储
- 索引压缩

- **布尔检索的概念**

- 在布尔检索中，文档被表示为**关键词的集合**。
- 所有的查询式都被表示为关键词的**布尔组合**。
  - 采用“与、或、非”关系加以连接
- 相关度计算
  - 一个文档当且仅当它能够满足布尔查询式时，才会将其检索出来。
  - 检索策略是**二值匹配**。



- **布尔检索的优缺点**

优点

- 查询简单，易于理解
- 使用布尔表达式，可以方便地控制查询结果
- 可通过扩展来包含更多功能

缺点

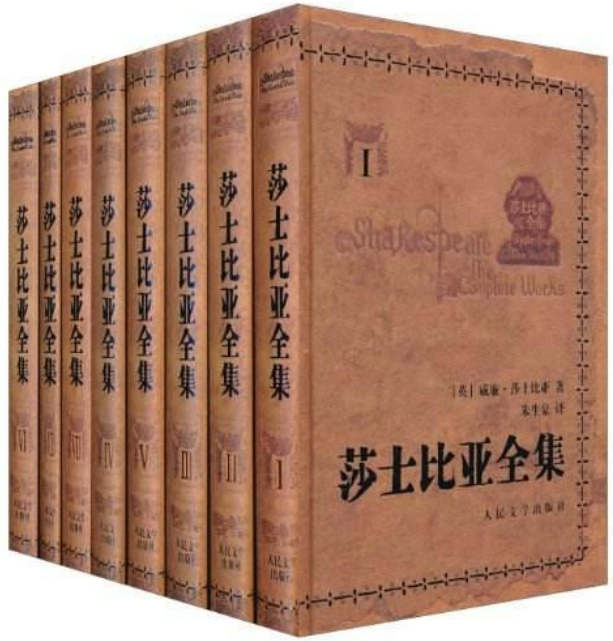
- 功能较弱，不支持部分匹配
- 所有匹配文档均返回，不考虑权重和排序
- 很难进行自动的相关性反馈

- **布尔检索的应用场景**

- 利用AND，OR或NOT等操作符，将词项连接起来
  - 操作符可以连续使用，如“信息 AND (检索 OR 挖掘) AND 教材”

- **一个布尔检索的实例**

- 《莎士比亚全集》中，哪些剧本包含Brutus和Caesar，但不包含Calpurnia?
  - 采用如下布尔表达式：Brutus AND Caesar AND NOT Calpurnia
- 一个解决的笨办法：从头到尾扫描所有剧本
  - 最大缺陷：速度超慢（尤其对于大型文档）
  - 处理NOT操作并不容易，需要遍历全文
  - 不支持检索结果的排序



- **另一种选择：关联矩阵**

- 通过采用非线性的扫描方式来解决，一种方法是事先给文档建立索引（Index）

	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

- **关联矩阵的查询实例**

- 关联矩阵的每一列都是0/1向量，每个0/1对应一个词项
  - 1代表包含这个词，0代表不包含
- 将给定的查询条件Brutus AND Caesar AND NOT Calpurnia转化为行向量运算
  - 取出三个行向量，对Calpurnia的行向量求补，最后按位进行与操作

- **110100 AND 110111 AND 101111 = 100100**

	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

- **关联矩阵的查询实例：结果**

- 剧本1：安东尼与克里奥佩特拉 (Antony and Cleopatra)，第三幕，第2场
  - 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.
- 剧本2：哈姆雷特 (Hamlet)，第三幕，第2场
  - Lord Polonius: I did enact Julius **Caesar** I was killed i' the Capitol; **Brutus** killed me.

- **关联矩阵的缺陷**

- 如果文档集很大……?
  - 假如有M篇文档，词汇表大小为N，那么矩阵规模为 M x N.
  - 在现实条件下，M和N的数量级都将达到百万甚至上亿的水平。
- 更为重要的是，矩阵中 1 的数量会微乎其微。
  - 高度稀疏的矩阵（可能小于1/500）
  - **只记录1的位置更为合理!**

本节目录

- 布尔检索
- **倒排索引**
  - **倒排表的构建与查询**
  - 倒排表的优化与扩展
- 索引存储
- 索引压缩

倒排索引的概念与意义

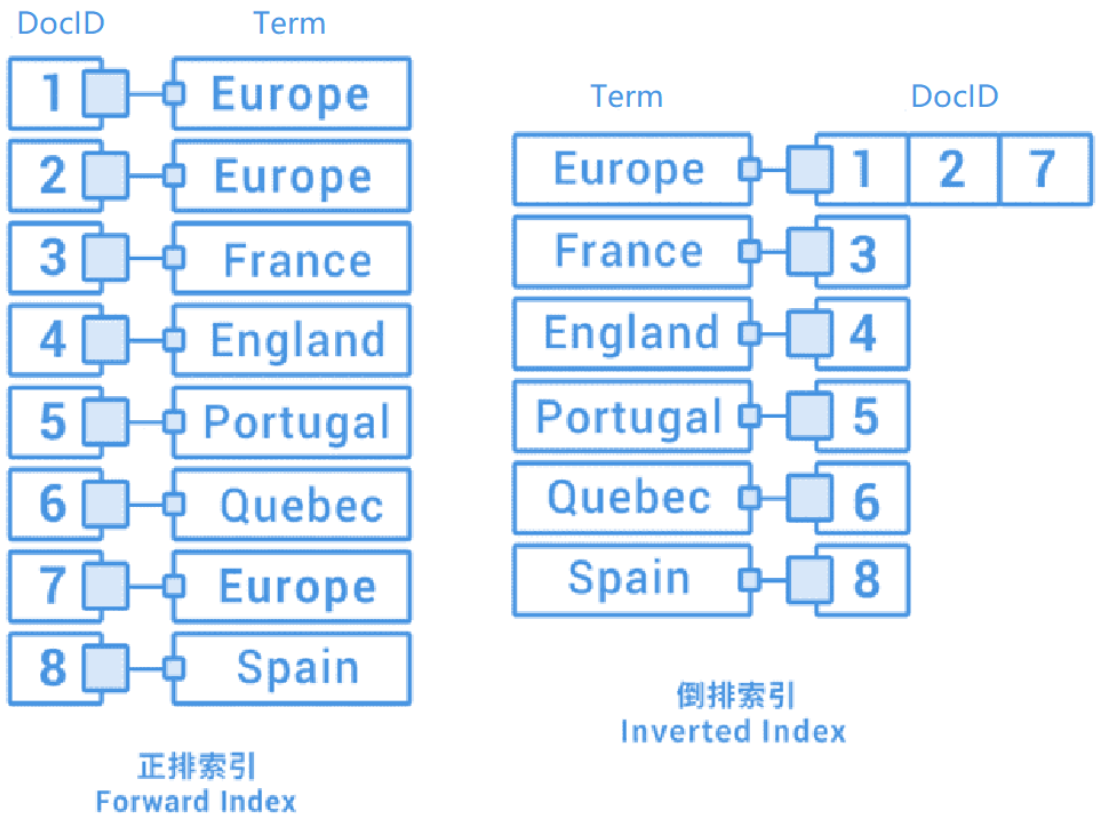
- 信息检索中非常流行的、基于词项的基础文本索引
- 主要包括以下两部分结构：
  - 词汇表（词典，Dictionary）：词项的集合
  - 倒排表（Posting List）：文档ID列表，列举词项在哪些文档中出现

Vocabulary		Postings List
term1	→	Document17, Document 45123
		.
		.
termN	→	Document991, Document123001

- **倒排索引的概念与意义**

- 正排索引VS倒排索引

	正排	倒排
关键索引（key）	Document	Term
优点	易维护	搜索快
缺点	搜索耗时长	构建、维护成本高



- **倒排索引的实例**

- 三篇简单文档的倒排表与检索实例
  - Yes, we got no bananas.
  - Johnny Appleseed planted apple seeds.
  - We like to eat, eat, eat apples and bananas.

Vocabulary		Postings List
yes	→	D1
we	→	D1, D3
got	→	D1
no	→	D1
bananas	→	D1, D3
Johnny	→	D2
Appleseed	→	D2
planted	→	D2
apple	→	D2, D3
seeds	→	D2
like	→	D3
to	→	D3
eat	→	D3
and	→	D3

倒排索引的实例：建立倒排表的流程

- 三步走的基本算法
  - 第一步：检索每篇文档，获得<词项，文档ID>对，并写入临时索引
    - Yes, we got no bananas.
    - Johnny Appleseed planted apple seeds.
    - We like to eat, eat, eat apples and bananas.

yes	→ D1
we	→ D1
got	→ D1
no	→ D1
bananas	→ D1

Johnny	→ D2
Appleseed	→ D2
planted	→ D2
apple	→ D2
seeds	→ D2

we	→ D3
like	→ D3
to	→ D3
eat	→ D3
eat	→ D3
eat	→ D3
apples	→ D3
and	→ D3
bananas	→ D3

- **倒排索引的实例：建立倒排表的流程**

- 三步走的基本算法
  - 第二步：对临时索引中的词项进行排序

yes	→ D1	we	→ D3
we	→ D1	like	→ D3
got	→ D1	to	→ D3
no	→ D1	eat	→ D3
Bananas	→ D1	eat	→ D3
Johnny	→ D2	eat	→ D3
Appleseed	→ D2	apples	→ D3
planted	→ D2	and	→ D3
apple	→ D2	bananas	→ D3
seeds	→ D2		

and	→ D3	Johnny	→ D2
apple	→ D2	like	→ D3
apple	→ D3	no	→ D1
Appleseed	→ D2	planted	→ D2
bananas	→ D1	seeds	→ D2
bananas	→ D3	to	→ D3
eat	→ D3	we	→ D1
eat	→ D3	we	→ D3
eat	→ D3	yes	→ D1
got	→ D1		

- **倒排索引的实例：建立倒排表的流程**

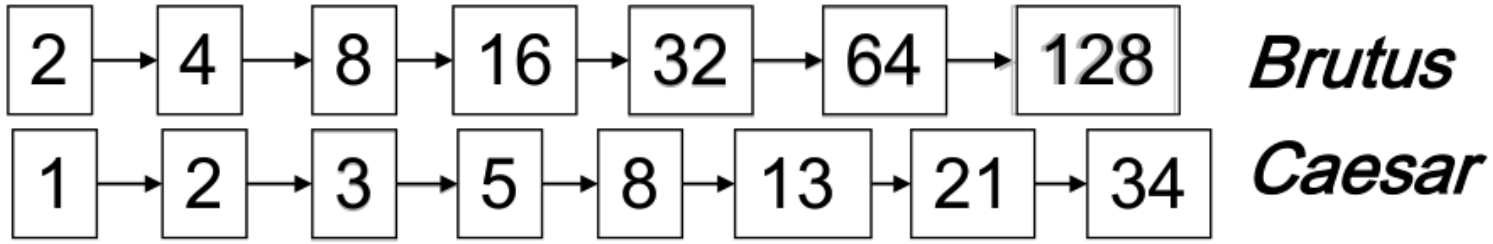
- 三步走的基本算法
  - 第三步：遍历临时索引，对于相同词项的文档ID进行合并

and	→ D3	Johnny	→ D2
apple	→ D2	like	→ D3
apple	→ D3	no	→ D1
Appleseed	→ D2	planted	→ D2
bananas	→ D1	seeds	→ D2
bananas	→ D3	to	→ D3
eat	→ D3	we	→ D1
eat	→ D3	we	→ D3
eat	→ D3	yes	→ D1
got	→ D1		

and	→ D3	no	→ D1
apple	→ D2, D3	planted	→ D2
Appleseed	→ D2	seeds	→ D2
bananas	→ D1, D3	to	→ D3
eat	→ D3	**we**	**→ D1, D3**
got	→ D1	yes	→ D1
Johnny	→ D2		
Like	→ D3		

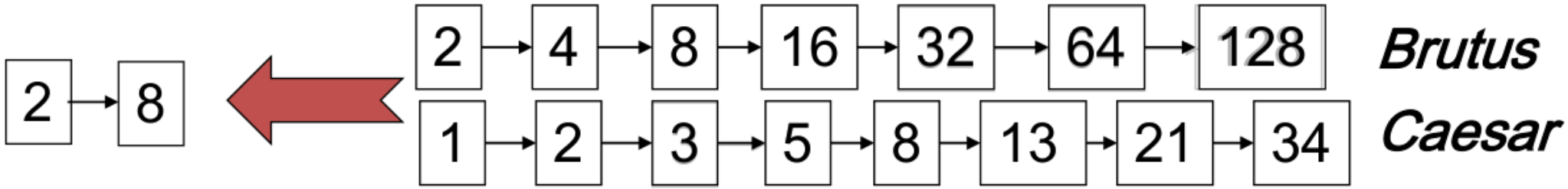
- **倒排索引的实例：基于倒排表的索引**

- 以莎翁的剧本检索为例，假如我们又双叒要查询Brutus AND Caesar
- 不妨假设两个词项的倒排表遵循以下形式：
  - Brutus：2，4，8，16，32，64，128……
  - Caesar：1，2，3，5，8，13，21，34……



- **倒排索引的实例：基于倒排表的索引**

- 基于倒排表的查询，本质上是倒排记录表的“合并”过程
- 同时扫描两个倒排表，所需时间与倒排记录的数量呈线性关系
  - 如果两个倒排表的长度分别为x和y，则合并共需O(x+y)次操作



- **倒排索引的索引算法**

```
INTERSECT(p1, p2)
  answer ← ⟨ ⟩
  while p1 ≠ NIL and p2 ≠ NIL
  do if docID(p1) = docID(p2)
        then ADD(answer, docID(p1))
             p1 ← next(p1)
             p2 ← next(p2)
        else if docID(p1) < docID(p2)
                then p1 ← next(p1)
                else p2 ← next(p2)
  return answer
```

动态索引问题

- 迄今为止，我们都假设文档集是静态的。但事实上，文档集通常不是静态的：
  - 文档会不断的加入进来
  - 文档也会被删除或者修改
- 这就意味着词典和倒排记录表需要修改：
  - 对于已在词典中的词项更新倒排记录
  - 将新的词项加入到词典中

https://www.youtube.com/watch?v=XiBjWoXrtCc

- **动态索引问题**

- 最简单的方法：主从索引
  - 维护一个大的主索引
  - 新文档信息存储在一个小的辅助索引中
  - 检索时同时遍历两个索引，并进行合并
- 如需删除操作，可利用一个新的无效位向量
  - 返回结果前，利用该向量过滤结果
- 定期将辅助索引合并到主索引中

https://www.youtube.com/watch?v=XiBjWoXrtCc

- **动态索引问题**

- 主从索引模式存在的问题
  - 频繁合并将导致很大的开销。
  - 合并过程效率很低。
    - 如果每个词项的倒排表单独形成一个文件，查询与合并将会较为简单，但此时压力转嫁到了文件读写上（因为有着过多的文件）。
- 现实中，往往在上述两种极端机制中取一个折中方案
  - 例如，对非常大的索引记录表进行切分，而对较短的索引记录表进行合并
    - 或者基于查询词项的常用性进行切分（需要对使用频率进行预判）

- **动态索引问题**

- 主从索引合并

```
LMergeAddToken(indexes, Z0, token)
  Z0 ← Merge(Z0, {token})
  if |Z0| = n
    then for i ← 0 to ∞
           do if Ii ∈ indexes          Ii 已存在
                then Zi+1 ← Merge(Ii, Zi)
                       (Zi+1 is a temporary index on disk.)
                     indexes ← indexes − {Ii}
                else Ii ← Zi      (Zi becomes the permanent index Ii.)
                     indexes ← indexes ∪ {Ii}
                     Break
           Z0 ← ∅
```

```
LogarithmicMerge()
  Z0 ← ∅      (Z0 is the in-memory index.)
  indexes ← ∅
  while true
  do LMergeAddToken(indexes, Z0, getNextToken())
```

本节目录

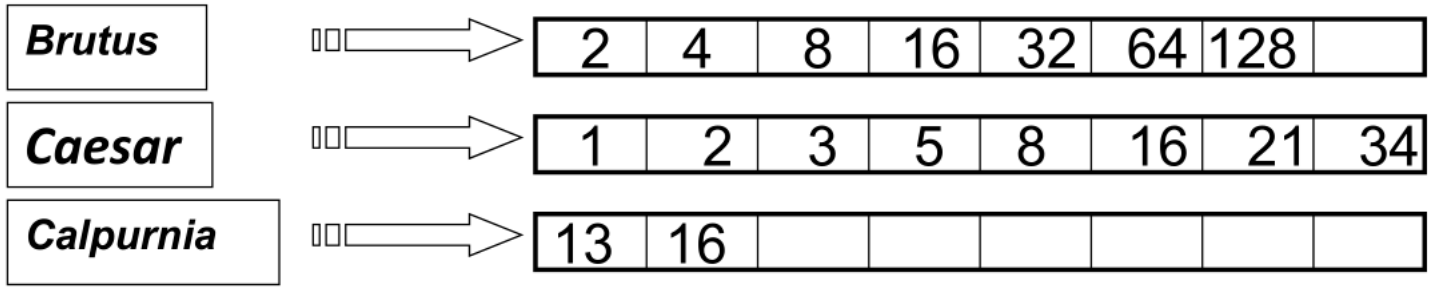
- 布尔检索
- **倒排索引**
  - 倒排表的构建与查询
  - **倒排表的优化与扩展**
- 索引存储
- 索引压缩

- **倒排索引的优化问题**

- 对于含有NOT操作的布尔查询，是否仍然能在O(x+y)的时间内完成?
  - 例如，Brutus AND NOT Caesar，如何处理?
- 进而，对于任意组合（例如，包含括号）的布尔查询，如何处理?
  - 例如，(Brutus OR Caesar) AND NOT (Antony OR Cleopatra)
  - 如何能够缩短查询的总时间开支?
- **核心问题：如何提升倒排索引的效率?**

倒排索引的优化问题

- 处理查询的最佳顺序是什么？
  - 对于使用AND连接的查询，其本质是倒排表的合并操作



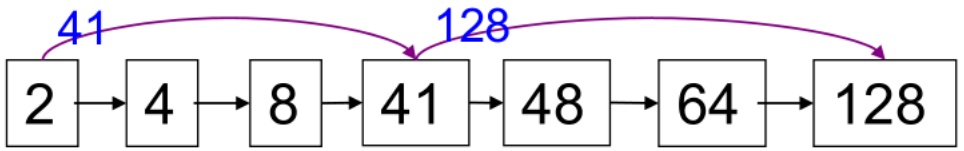
  - 按照文档频率的顺序进行处理
    - 先处理文档频率小的，再处理大的
    - 在上例中，我们采用(Calpurnia AND Brutus) AND Caesar的顺序处理

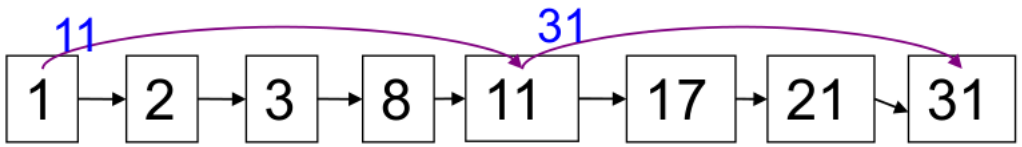
- **倒排索引的优化问题**

- 更一般的优化问题：任意组合的布尔查询
  - 例如：(Brutus OR Caesar) AND (Antony OR Cleopatra)
  - 同样，按照文档频率的顺序进行处理
    - 首先，获得所有词项的文档频率
    - 其次，保守地估计出每个OR操作后的结果大小
      - 考虑x+y的最坏情况
    - 最后，按照结果从小到大的顺序执行AND

- **倒排表合并中的优化问题**

- 更进一步：倒排表的合并需要O(x+y)次，能否做得更好?

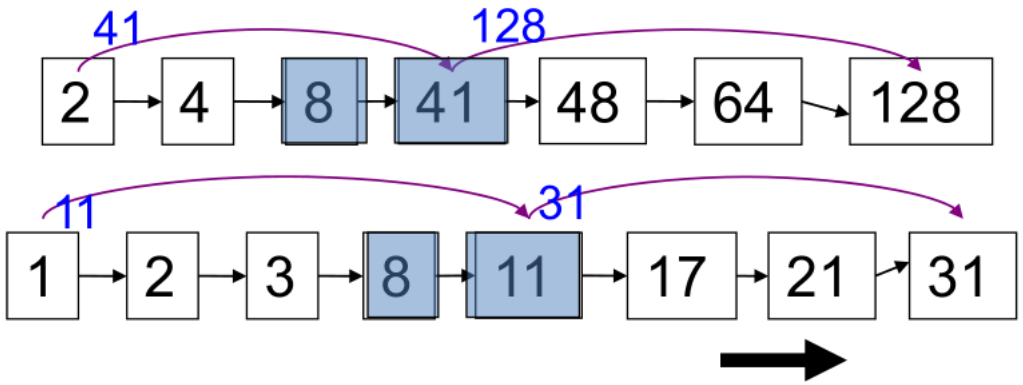




- 通过设置跳表指针跳过部分文档，从而实现快速合并
  - 如何利用跳表指针实现快速合并?
  - 在什么位置设置跳表指针?

- **倒排表合并中的优化问题**

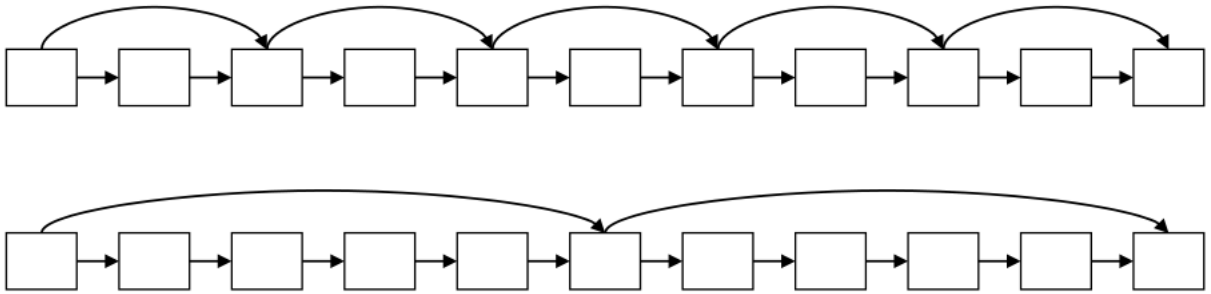
- 带有跳表指针的查询处理过程



- 首先，通过遍历发现了共同的记录8，继续移动指针
- 其次，表2在11的位置，我们发现跳表指针31小于表1的下一个数41
- 因此，我们直接将表2跳到31，而跳过其中的17、21两个数

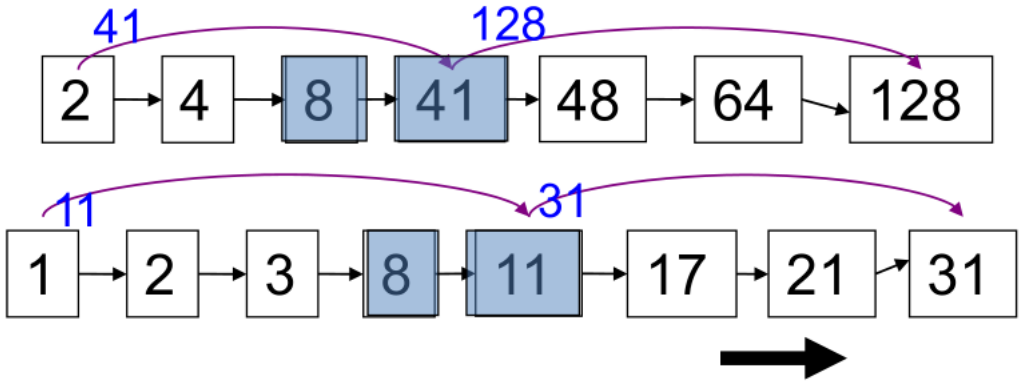
- **倒排表合并中的优化问题**

- 以何种策略设置跳表指针?
  - 设置较多的指针+较短的步长→更多的跳跃机会。
    - 相应的，需要耗费更多的存储空间
  - 设置较少的指针+较长的步长→更少的指针比较次数
    - 存储空间消耗更少，但跳跃机会也更少



• 倒排表合并中的优化问题

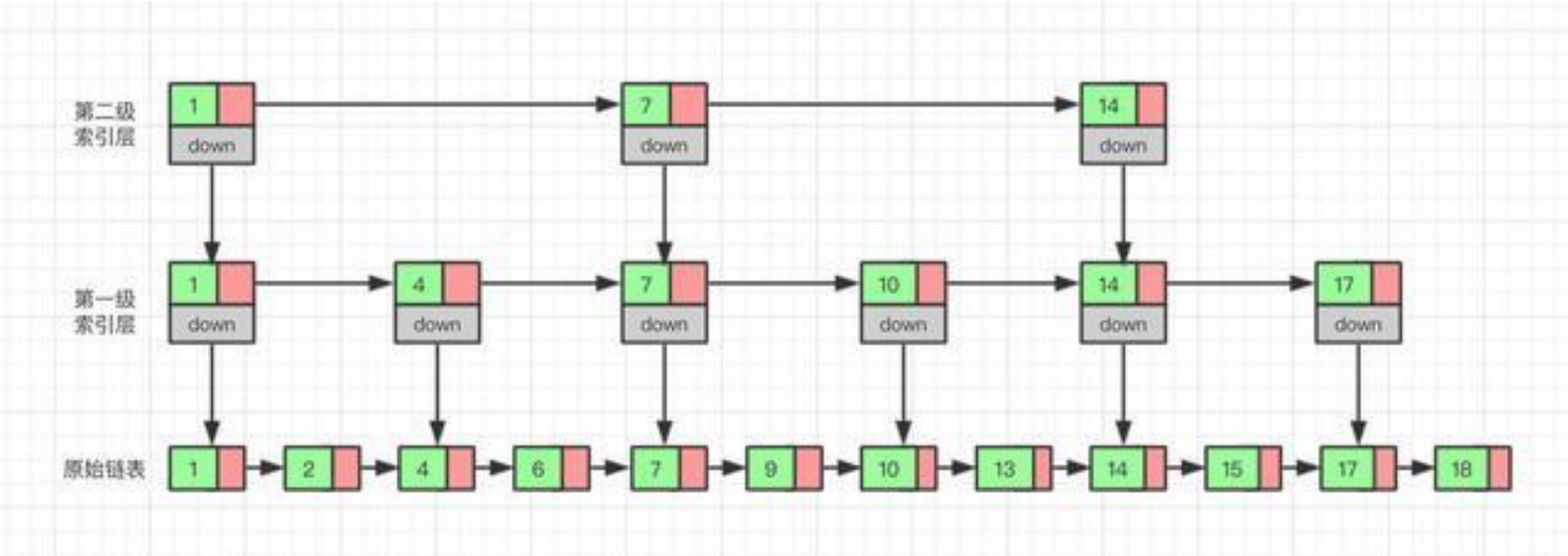
- 一个简单的启发式策略：如果倒排表长度为L，则间隔$\sqrt{L}$均匀放置跳表指针
  - 该策略没有考虑查询词项的分布，未必导致结果优化。
  - 同时，索引的动态变化也会影响跳表指针的设置
    - 如果索引相对固定，建立有效的跳表指针就相对容易
    - 反之，经常更新的索引很难建立合适的跳表指针



- **倒排表合并中的优化问题**

- 进阶：多层跳表指针
  - 多层跳表指针的推进与单层类似，但高层节点还会多一个Down指针（指向下一层）
    - 如果当前处在第K层，发现对应数值y处在当前节点x_t和下一节点x_t+1之间，则进入下一层（即K-1层）进行更细粒度的比对

该过程与二分查找类似!



- **短语查询的需求**

- 用户希望将类似“stanford university”的查询中的两个词看成是一个整体
  - 类似“I want to university at stanford”这样的文档是不能够满足用户需求的。
  - 大部分的搜索引擎都支持双引号的短语查询，这种语法很容易理解并使用。然而，有很多查询在输入时没有加双引号，其实都是隐式的短语查询。

- **第一种解决方案：二元词索引**

- 将文档中每个连续词对看成一个短语
  - 例如，文本“Friends, Romans, Countrymen”将生成如下的二元连续词对：
    - Friend Roman
    - Roman Countrymen

    以上的每一个二元词对都将作为词典中的词项。
  - 经过上述的处理，可以构建面向二元词的倒排表，并处理两个词（或多个词）构成的短语查询。

- **第一种解决方案：二元词索引**

- 更长的短语查询：可以分成多个短查询来处理
  - 例如，文本“stanford university palo alto”
    - 将分解成如下的二元词对布尔查询：
      - stanford university AND university palo AND palo alto
    - 或采用更长的多元词索引加以解决
  - 如果采用二元词索引拼接的方式，对于该布尔查询返回的文档，我们**不能确定**其中是否真正包含最原始的四词短语。

- **第二种解决方案：位置信息索引**

- 二元词（或多元词）索引最大的问题：词汇表迅速增长
- 在记录词项的同时，记录它们在文档中出现的位置，可以达成更广泛的查询
  - 在这种索引中，对每个词项，采用以下方式存储其倒排表记录：

〈词项，词项频率；
文档1：位置1，位置2……；
文档2：位置1，位置2……；
……〉

<***be***: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>

- **第二种解决方案：位置信息索引**

- 位置信息检索实例

<*be*: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>

Which of docs 1,2,4,5 could contain "***to be or not to be***"?

- 对于短语查询，仍采用合并算法（AND），查找符合的文档。
- 不只是简单地判断两个词是否出现在同一文档中，还需要检查他们出现的位置情况是否符合要求。

- **第二种解决方案：位置信息索引**

- 位置信息检索实例

  - 例子：

  查询词：“to be or not to be”

  倒排表：

  - *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; . . .

1. 考虑to和be的倒排表的合并，查找同时包含to和be的文档
2. 检查表中，看看是否某个be的前面的一个位置上正好出现to

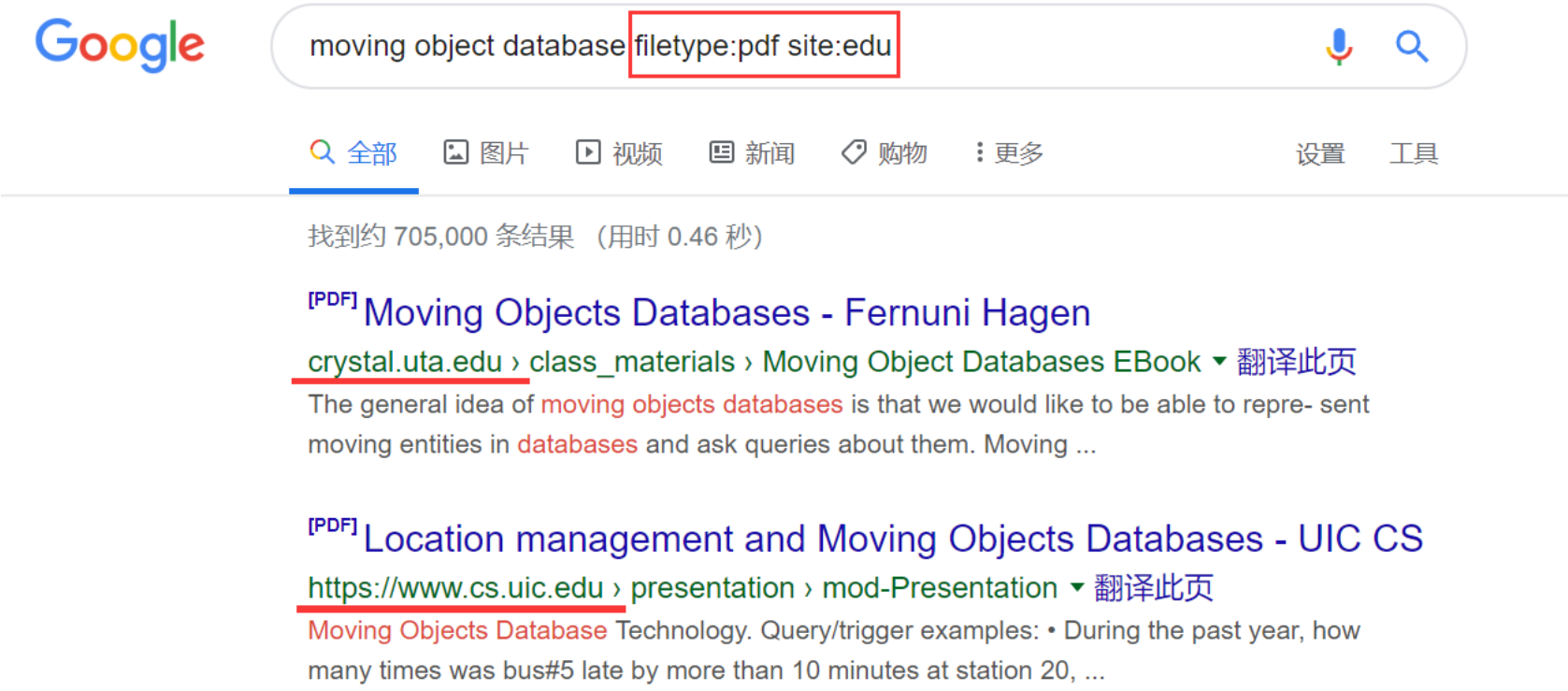
- 更为重要的是，位置信息索引能够用于邻近搜索（例如，间隔k个词）

- **倒排表的扩展性问题**

- 除了文档ID和位置，搜索引擎往往在倒排表中加入更多元素
  - 例如：**词项频率（Term Frequency）**、词项类型等
- 同时，除了基本的词查询之外，倒排表还可能面临更多需求
  - 文档范围：例如文档来源，site:www.ustc.edu.cn
  - 文档类型：例如搜索引擎常用的filetype:PDF
  - 文档属性：作者是“D.Manning”的文档

Vocabulary		Postings List
yes (docs=1)	→	D1 (freq=1)
we (docs=2)	→	D1 (freq=1), D3 (freq=1)
…		
eat (docs=1)	→	D3 (freq=3)
…		

- **倒排表的扩展性问题：实例**



本节目录

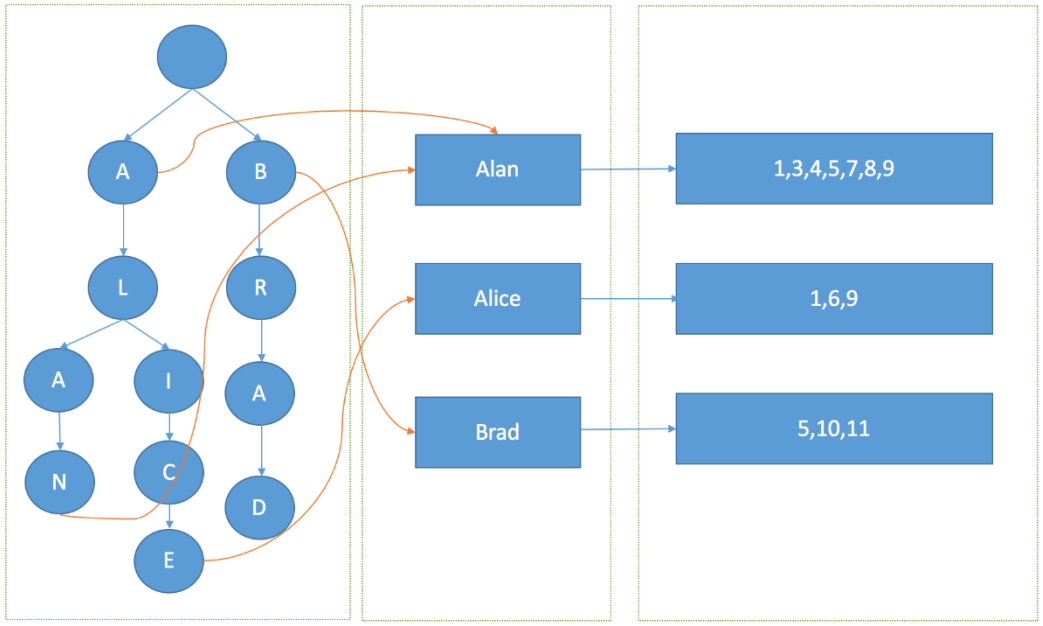
- 布尔检索
- 倒排索引
  - 倒排表的构建与查询
  - 倒排表的优化与扩展
- **索引存储**
- 索引压缩

- **倒排索引的存储问题**

- 一种常用方式：词典与倒排表一起存储
  - 便于同时读取，但文档规模大时将导致索引过大，影响性能
- 另一种常用方式：两者分开存储
  - 词典与倒排表分别存储为不同的文件，通过页指针关联
    - 倒排表文件也可采用分布存储的方式
  - 优点：性能大幅提升
    - 词典可以常驻内存，至少常驻一部分（例如主索引）
    - 可以支持并行、分布式查询

- **倒排索引的存储问题**

- 常见的词汇表存储结构
  - 顺序存储
  - 哈希存储
  - B/B+树
  - Trie树

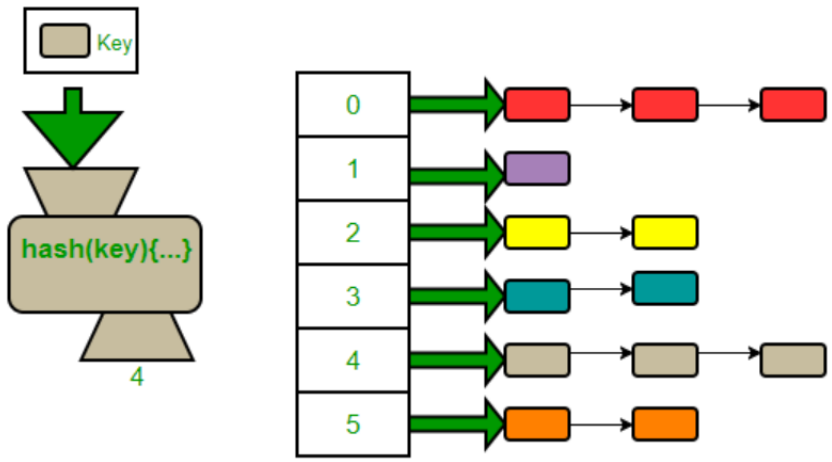


- **最基本的存储方式：顺序存储**

- 词汇表的顺序排列方式
  - 把词汇表按照字典顺序进行排列（查询的前提）
  - 词汇表的查找采用二分查找法
- 优点：简单粗暴
- 缺点：效率一般
  - 索引构建的效率一般（文档插入需要反复调用查找和排序）
  - 索引检索的效率也一般
    - 二分查找复杂度为O(LogN)，与词汇数量N有关（通常海量词汇）

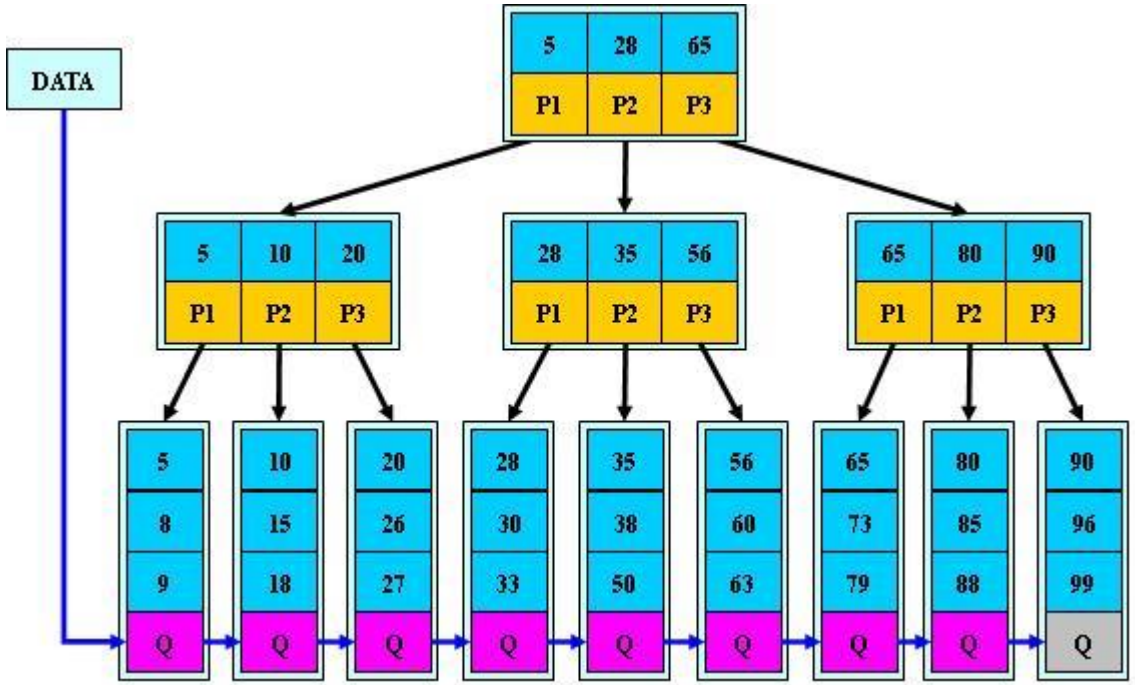
改进存储：哈希存储

- 对词汇表进行哈希
  - 根据给定的词项，散列成一个整数
  - 用该整数作为词项的访问地址
- 优点：实现简单、检索速度快，理论时间O(1)
- 缺点：
  - 当冲突过多时效率会下降
    - 例如，对姓名使用简称进行哈希，则姚明≈杨幂 = YM
  - 想要避免冲突，哈希值的取值空间就会带来巨大的存储压力
  - 关键在于找到一个好的散列函数



- **B/B+树存储**

- 对词汇表进行B/B+树存储
  - 多叉平衡有序树
- 优点：
  - 性能好且稳定，查找次数 = 层数
- 缺点：
  - 维护代价较高
  - 实现相对复杂

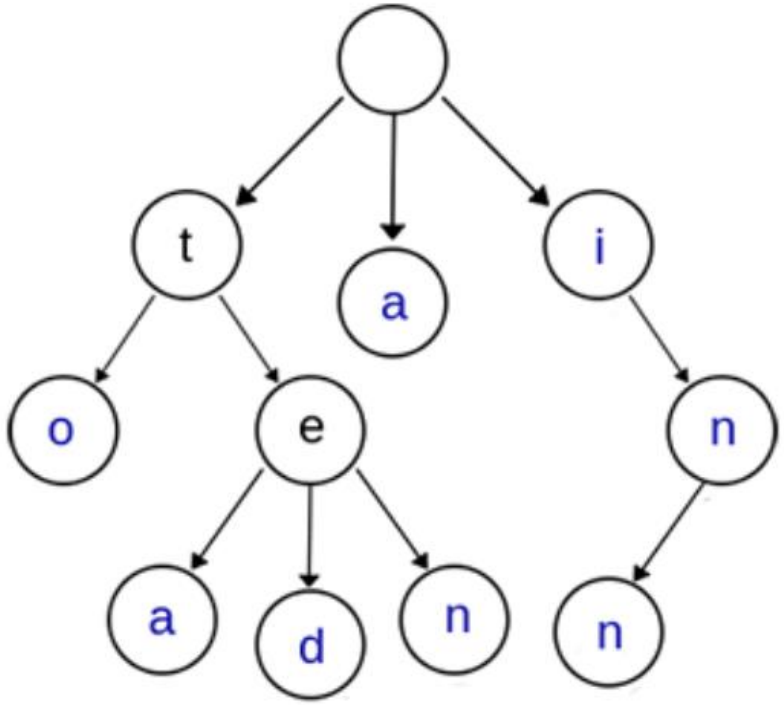


- **Trie树**

- Trie树，又称前缀树，利用字符串的公共前缀来节约存储空间
  - Trie树是一种用于快速检索的多叉树结构
  - Trie树把要查找的关键词看作一个字符序列
  - 根节点不包含字符，除根节点外每一个节点都只包含一个字符
  - 从根节点到某一节点，经过的字符连接起来即为该节点对应的字符串
  - 每个节点的所有子节点包含的字符都不相同
- 查找时间只与词的长度有关，而与词典中词的个数无关。
  - 当词表较大时，才能体现出速度的优势

- **Trie树**

- 例如，以下Trie树对应词典单词：
  - t、a 、i 、to 、te 、in 、tea 、ted、ten、inn



- **Trie树**

- Trie树的优缺点：
  - 优点：效率高
    - 查找效率高，与词表长度无关
    - 索引的插入，合并速度快
  - 缺点：所需空间较大，本质是“以空间换时间”
    - 如果是完全m叉树，节点数指数级增长
    - Trie树虽然不是完全m叉树，但所需空间仍然很大，尤其当词项公共前缀较少时

本节目录

- 布尔检索
- 倒排索引
  - 倒排表的构建与查询
  - 倒排表的优化与扩展
- 索引存储
- **索引压缩**

- **索引压缩的意义**

- 为什么要压缩索引?
  - 节省磁盘空间（￥），提高效率（内存利用率或数据传输速度）
  - 前提：快速的解压缩算法。目前的启发式算法效率都比较高。
- 对于词典而言，压缩的意义：
  - 压缩得足够小，可以直接放入内存中，提升效率
- 对于倒排记录表而言，压缩的意义：
  - 减少所需的磁盘空间，可以更多移入内存
  - 减少从磁盘读取倒排表所需的时间

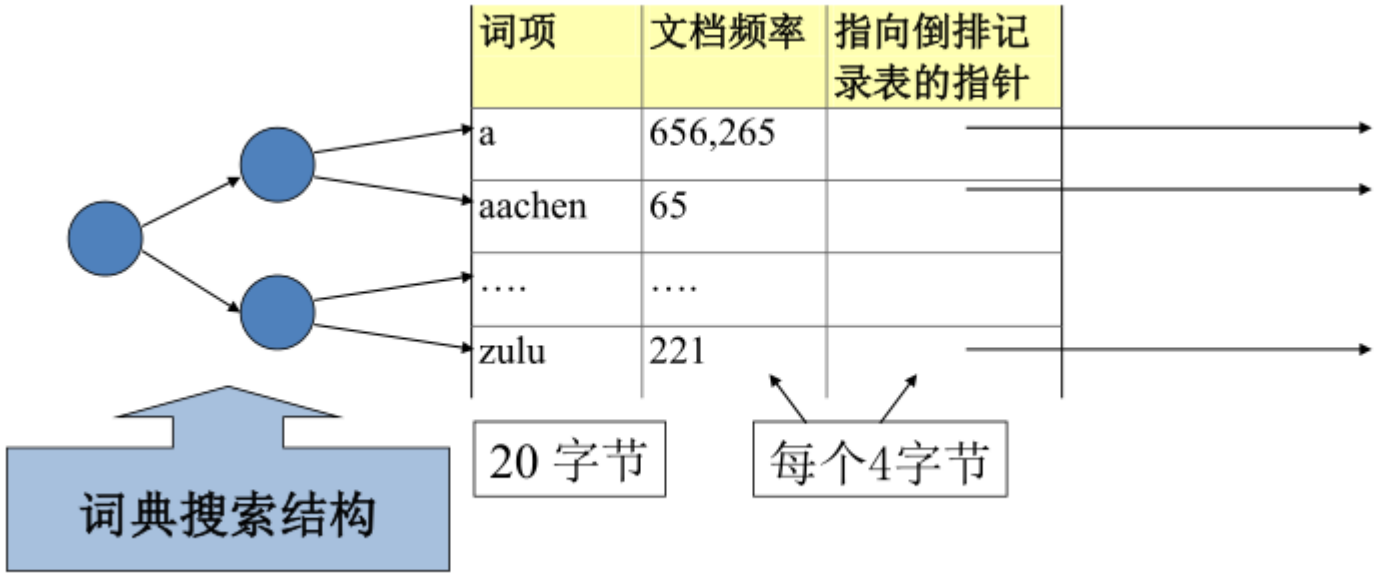
- **两种索引压缩策略**

- 从词典和倒排表两个维度入手，实现索引的压缩。
  - 以Reuters-RCV1语料库为例，比较不同压缩手段下的空间需求变化。

符号	含义	值
N	文档总数	800,000
L	每篇文档的平均词条数目	200
M	词项总数	400,000
	每个词条的平均字节数（含空格和标点符号）	6
	每个词条的平均字节数（不含空格和标点符号）	4.5
	每个词项的平均字节数	7.5
	倒排记录总数	100,000,000

- **词典的基本存储方式：定长存储**

- 在词项总数为400K的情况下，考虑每个词项占28字节，一共需要11.2 MB
  - 词项本身设置定长为20字节，另外需要记录文档频率和指向倒排表。

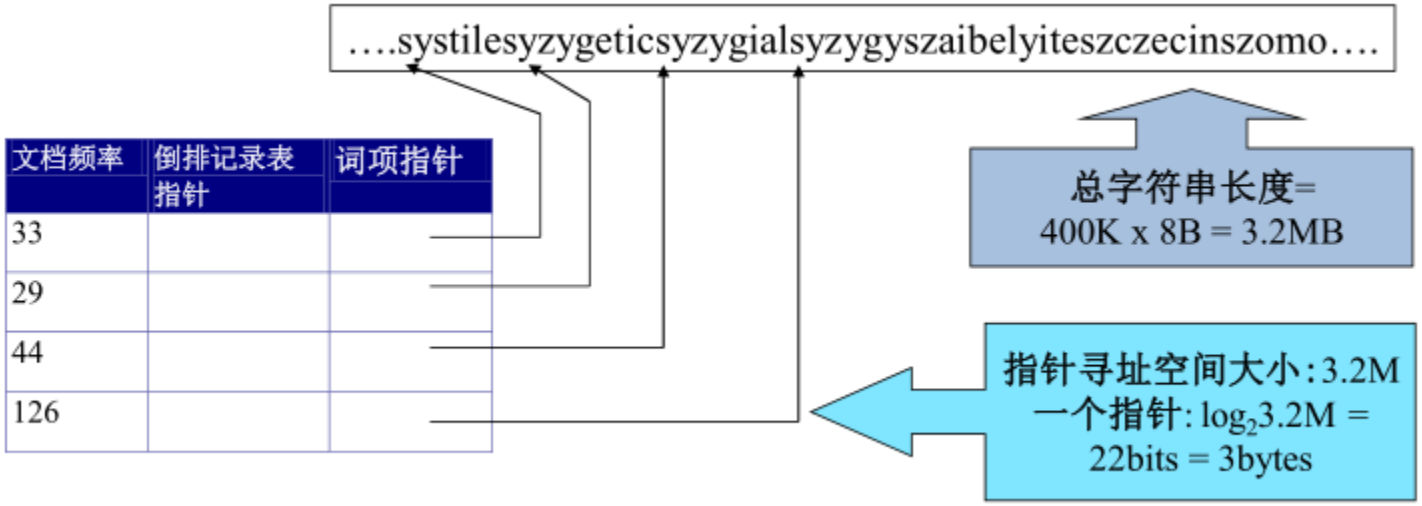


- **定长存储将造成空间浪费**

- 为每次词项设置定长为20字节，将导致极大的空间浪费
  - 书面英文中单词的平均长度为4.5个字符
  - 英文中平均的词典词项长度为8个字符
    - 部分短字符作为停用词被删去
    - 即使如此，仍然会造成12个字节的空间浪费（采用ASCII编码）
  - 糟糕的是，即使做出这样的让步，仍然有些超级长词无法被存储
    - E.g., Supercalifragilisticexpialidocious （奇妙的，难以置信的）

- **压缩词项列表：将词典视作单一字符串**

- Dictionary-as-a-String，词项之间用指针分割
  - 指向下一个词项的指针同时也标识着当前词项的结束
  - 期望节省60%的词典空间：(20-8)/20×100% = 60%

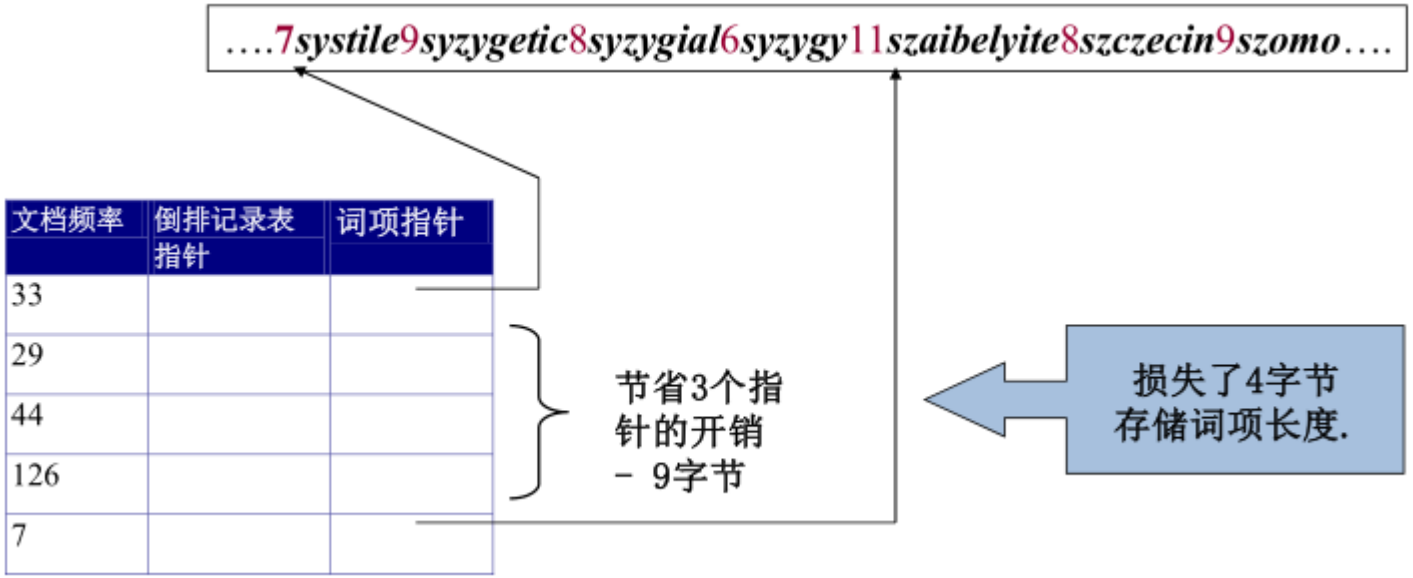


- **字符串词典的空间大小**

- 每个词项平均总计占用19个字节，而不是原先的28个字节
  - 首先，在词项字符串中，每个词项平均长度8个字节
  - 其次，词项文档频率与倒排表指针各4字节不变
  - 最后，词项指针约3字节
    - 字符总长度约为400K*8=3.2M，用大约22bit≈3B长度可以标记
  - 8+4+4+3=19，400K词项一共仅需7.6MB（定长存储时为11.2MB）

- **进一步压缩：按块存储（Blocking）**

- 单一字符串在词项指针上需要占用较多额外空间
- 通过为每k个词项存储一个指针，来减少指针的总数量
  - 需要**额外1个字节**用于表示词项长度。
  - 例如，下图中的例子为k=4



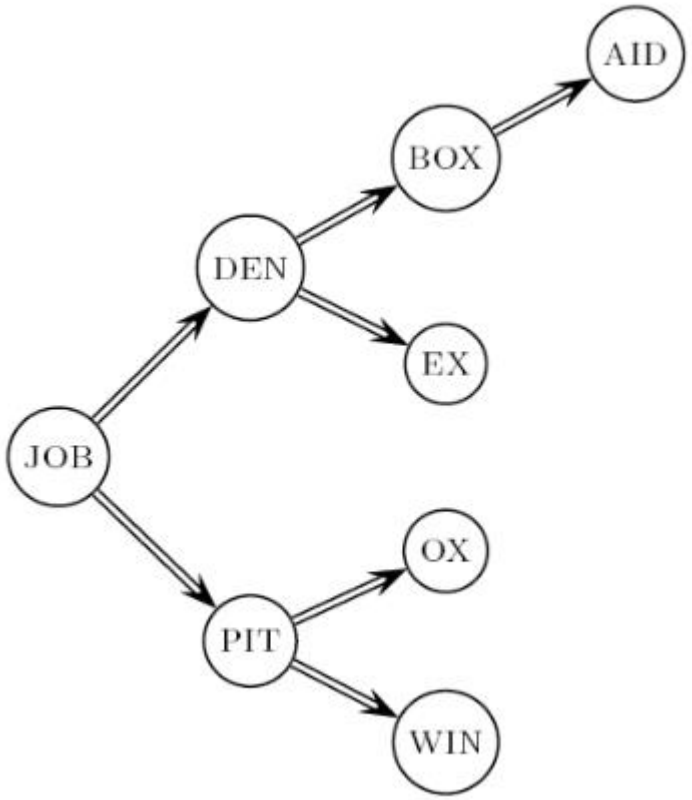
- **进一步压缩：按块存储（Blocking）**

- 按块存储通过牺牲少量存储词项长度，可以节省更多的指针开支
- 例如，当块的大小k=4时
  - 不采用按块存储时，每个指针花费3字节，共需12字节
  - 采用按块存储时，只需要花费3+4*1=7字节
- 由此，原先的存储空间可以进一步降低到7.1MB

- 讨论：k增大时，开支进一步降低，为什么不选取更大的k?

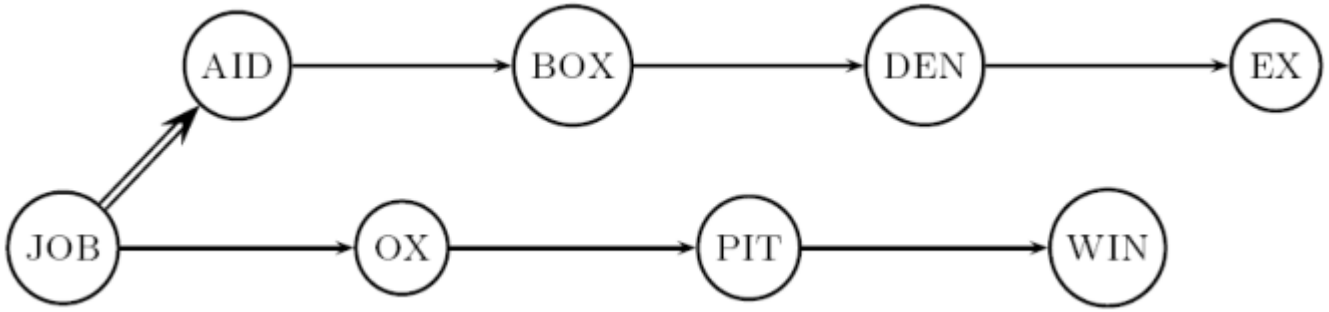
- **按块存储在搜索上的问题**

- 未压缩词典的搜索，标准二分法
  - 假设词典中每个词项被查询的概率相同（实际上并非如此）
  - 则平均比较次数为：
    - (1+2*2+4*3+4)/8=2.625次



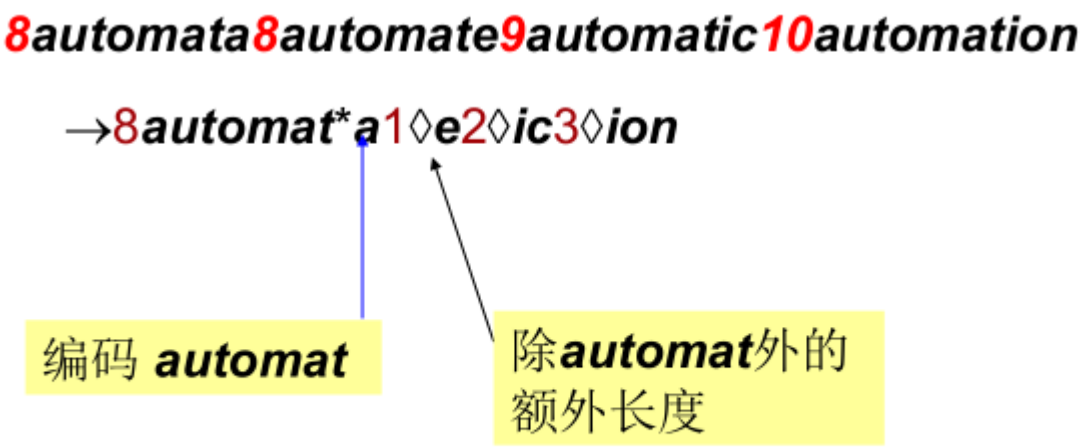
- **按块存储在搜索上的问题**

- 采用按块存储后，二分查找只能在块外进行
  - 块内采用线性查找方式。
- 当块的大小k=4，平均比较次数为：(1+2*2+2*3+2*4+5)/8 = 3步
  - 显然，随着k上升，线性查找部分增多，效率更低



- **另一种改进：前端编码（Front Coding）**

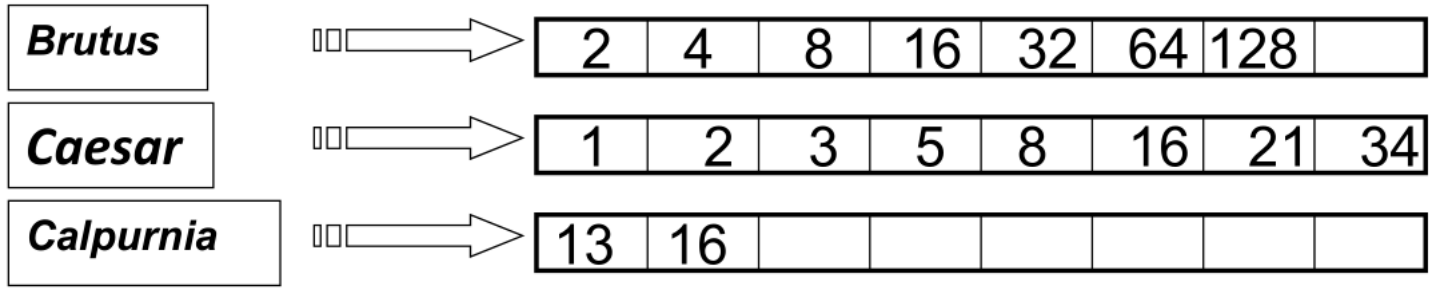
- 按照词典顺序排列的连续词项之间，往往具有公共的前缀
  - 使用特殊字符表示前缀使用，如下图的◊
  - 对于RCV1语料库而言，前端编码可以将按块存储所需的7.1MB降至5.9MB



类似一般的字符串压缩方法

- **倒排表存储的问题所在**

- 倒排表所需的空间远远大于词典本身
- 最迫切的需求在于如何紧密地存储每一个倒排表，尤其是文档ID
  - 如果使用4字节整数来表示文档ID，每个文档ID需要32bit
  - 对于RCV1语料库而言，800K文档意味着至少需要$\log_2 800000 \approx 20$ bits
  - **如何用远小于20bit来表示每个文档ID?**



- **线索：文档集中词项的分布情况——Zipf定律**

- 只有很少一些非常高频的词项，其它绝大部分都是很生僻的词项。
- **Zipf 定律**：排名第 **i** 多的词项的文档集频率与 **1/i** 成正比
  - 另一种表述方式：任意一个词项，其频度（Frequency）的排名（Rank）和频度的乘积大致是一个常数
  - $Cf_i * i \approx K$， Cf_i为文档集频率（排名第 i 的），K为归一化常数

Tips: Zipf 定律是 Zipf 在 1949 年的一本关于人类定位的最小作用原理的书中首先提出的，其中最令人难忘的例子是在人类语言中，如果以单词出现的频次将所有单词排序，用横坐标表示序号，纵坐标表示对应的频次，可以得到一条幂函数曲线。这个定律被发现适用于大量复杂系统。

- **倒排表存储的两种相反需求**

- 词项的频率巨大差异，决定了对于倒排表存储的不同需求
  - hydrochlorofluorocarbons这种词项可能成百上千万个文档中才出现一次
    - 采用$\log_2 800000 \approx 20$ bits来记录这一倒排记录，可以满足需求
  - The这种词项（如果没有作为停用词删除）可能每个文档都会出现
    - 类似这种情况，采用20bits记录太浪费了
    - 1bit的提示符（e.g., 0/1——某篇文档有或没有）即可满足

- **规律观察：采用间距代替文档ID**

- 一个基本的道理：间距的数值必然小于文档ID的数值
  - 例如，词项Computer在倒排表中的记录为：33,47,154,159,202…
  - 如果采用间距存储，该表可改为：33,14,107,5,43…
- 期望：绝大多数间距存储空间都远小于20bits

	encoding	postings list					
THE	docIDs	...	283042	283043	283044	283045	...
	gaps			1	1	1	...
COMPUTER	docIDs	...	283047	283154	283159	283202	...
	gaps			107	5	43	...
ARACHNOCENTRIC	docIDs	252000	500100				
	gaps	252000	248100				

- **再进一步：可变长度编码**

- 我们的需求：对于一个间距值G，想用最少的所需字节来表示它
- 关键问题：需要利用整数个字节来对每个间距编码
  - 这需要一个可变长度编码，对小数字使用短码来实现这一点
- 可变长度编码的基本流程大致如下：
  - 先存储G，并分配1bit作为延续位
  - 如果G<128，则采用第一位延续位为1 + 7位有效二进制编码的格式
  - 如果G>=128，则先对低阶的7位编码，然后采取相同算法对高阶位进行编码。最后一个字节（8bit）的延续位为1，其他字节延续位为0.

- **倒排表存储：可变长度编码**

- 可变长度编码的实例
- 例如，5的二进制为101，加上延续位为10000101.
- 214577的二进制为1101/0001100/0110001，因此拆分为3个字节。
- 相比于4字节整数，可变字节码在小数字上的短码可以节省更多空间。

文档ID	824	829	215406
间距		5	214577
VB 编码	00000110 10111000	10000101	00001101 00001100 10110001

倒排索引以一连串字节的形式存储

00000110101110001000010100001101000011001011000 1

对一个小的间距(5)，VB编码使用了一整个字节

本章小结

网页索引

- 布尔检索与关联矩阵
- 倒排索引
  - 倒排表构建、倒排检索、动态索引
  - 检索优化、倒排表的扩展
- 索引的存储
- 索引的压缩