

现代信息检索

Modern Information Retrieval

第3讲 索引压缩

Index compression

提纲

- ① 上一讲回顾
- ② 压缩
- ③ 词项统计量
- ④ 词典压缩
- ⑤ 倒排记录表压缩

提纲

- ① 上一讲回顾
- ③ 压缩
- ③ 词项统计量
- ③ 词典压缩
- ③ 倒排记录表压缩

基于块的排序索引构建算法BSBI

合并后的倒排记录表

待合并的倒排记录表

Term id	Doc id	Term id	Doc id
1	d1, d3	1	d6, d7
2	d1, d2, d4	2	d8, d9
3	d5	5	d10
4	d1, d2, d3, d5	6	d8



Term id	Doc id
1	d1, d3, d6, d7
2	d1, d2, d4, d8, d9
3	d10
4	d8
5	d5
6	d1, d2, d3, d5

合并过程基本不占用内存，但是需要维护一个全局词典

词典： 维护一张词项到整型词项ID的映射表

待合并的倒排记录表： 只包含整型ID，没有字符串

全局词典

term	term id	term	term id
brutus	1	with	4
caesar	2	julius	5
noble	3	killed	6

内存式单遍扫描索引构建算法SPIMI

- 关键思想 1：对每个块都产生一个独立的词典 - 不需要在块之间进行term-termID的映射
- 关键思想2：对倒排记录表不排序，按照他们出现的先后顺序排列
- 基础上述思想可以对每个块生成一个完整的倒排索引
- 这些独立的索引最后合并一个大索引

SPIMI-Invert算法

```
SPIMI-INVERT(token_stream)
1  output_file ← NEWFILE()
2  dictionary ← NEWHASH()
3  while (free memory available)
4  do token ← next(token_stream)
5      if term(token) ∉ dictionary
6          then postings_list ← ADDTODICTIONARY(dictionary, term(token))
7          else postings_list ← GETPOSTINGSLIST(dictionary, term(token))
8          if full(postings_list)
9              then postings_list ← DOUBLEPOSTINGSLIST(dictionary, term(token))
10         ADDTOPOSTINGSLIST(postings_list, docID(token))
11 sorted_terms ← SORTTERMS(dictionary)
12 WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13 return output_file
```

本讲内容

对每个词项 t , 保存所有包含 t 的 文档列表



- 信息检索中进行压缩的动机
- 词项统计量：词项在整个文档集中如何分布？
- 倒排索引中词典部分如何压缩？
- 倒排索引中倒排记录表部分如何压缩？

提纲

- ① 上一讲回顾
- ② 压缩
- ③ 词项统计量
- ④ 词典压缩
- ⑤ 倒排记录表压缩

什么是压缩？

- 将长编码串用短编码串来代替
 - 11111111111111111111 → 18个1

为什么要压缩? (一般意义上而言)

- 减少磁盘空间 (节省开销)
- 增加内存存储内容 (加快速度)
- 加快从磁盘到内存的数据传输速度 (同样加快速度)
 - [读压缩数据到内存+在内存中解压]比直接读入未压缩数据要快很多
 - 前提: 解压速度很快
- 本讲我们介绍的解压算法的速度都很快

为什么在IR中需要压缩?

- 占用更少的硬盘空间
 - 更经济，节省空间
- 将更多数据载入内存
 - 加快处理速度（内存中读写很快）
- 减少从磁盘读入内存的时间
 - 注意: 大型搜索引擎将相当比例的倒排记录表都放入内存
- IR中压缩的两个基本要求
 - （通常是）无损压缩
 - 随机访问（Random Access）
- 接下来，将介绍词典压缩和倒排记录表压缩的多种机制
 - 压缩的一个基本问题：对齐。即不同压缩单元之间的分界标识

有损(Lossy) vs. 无损(Lossless)压缩

- **有损压缩:** 丢弃一些信息
- 前面讲到的很多常用的预处理步骤可以看成是有损压缩:
 - 统一小写,去除停用词, Porter词干还原, 去掉数字
- **无损压缩:** 所有信息都保留
 - 索引压缩中通常都使用无损压缩

提纲

- ① 上一讲回顾
- ② 压缩
- ③ 词项统计量**
- ④ 词典压缩
- ⑤ 倒排记录表压缩

词典压缩和倒排记录表压缩

- 词典压缩中词典的大小即词汇表的大小是关键
 - 能否预测词典的大小？
- 倒排记录表压缩中词项的分布情况是关键
 - 能否对词项的分布进行估计？
- 引入词项统计量对上述进行估计，引出两个经验法则

对文档集建模： Reuters RCV1

<i>N</i>	文档数目	800,000
<i>L</i>	每篇文档的词条数目	200
<i>M</i>	词项数目(= 词类数目)	400,000
	每个词条的字节数 (含空格和标点)	6
	每个词条的字节数 (不含空格和标点)	4.5
	每个词项的字节数	7.5
<i>T</i>	无位置信息索引中的倒排记录数目	100,000,000

预处理的效果

	不同词项			无位置信息倒排记录			词 条 ^①		
	数目	Δ%	T%	数目	Δ%	T%	数目	Δ%	T%
未过滤	484 494			109 971 179			197 879 290		
无数字	473 723	-2	-2	100 680 242	-8	-8	179 158 204	-9	-9
大小写转换	391 523	-17	-19	96 969 056	-3	-12	179 158 204	-0	-9
30个停用词	391 493	-0	-19	83 390 443	-14	-24	121 857 825	-31	-38
150个停用词	391 373	-0	-19	67 001 847	-30	-39	94 516 599	-47	-52
词干还原	322 383	-17	-33	63 812 300	-4	-42	94 516 599	-0	-52

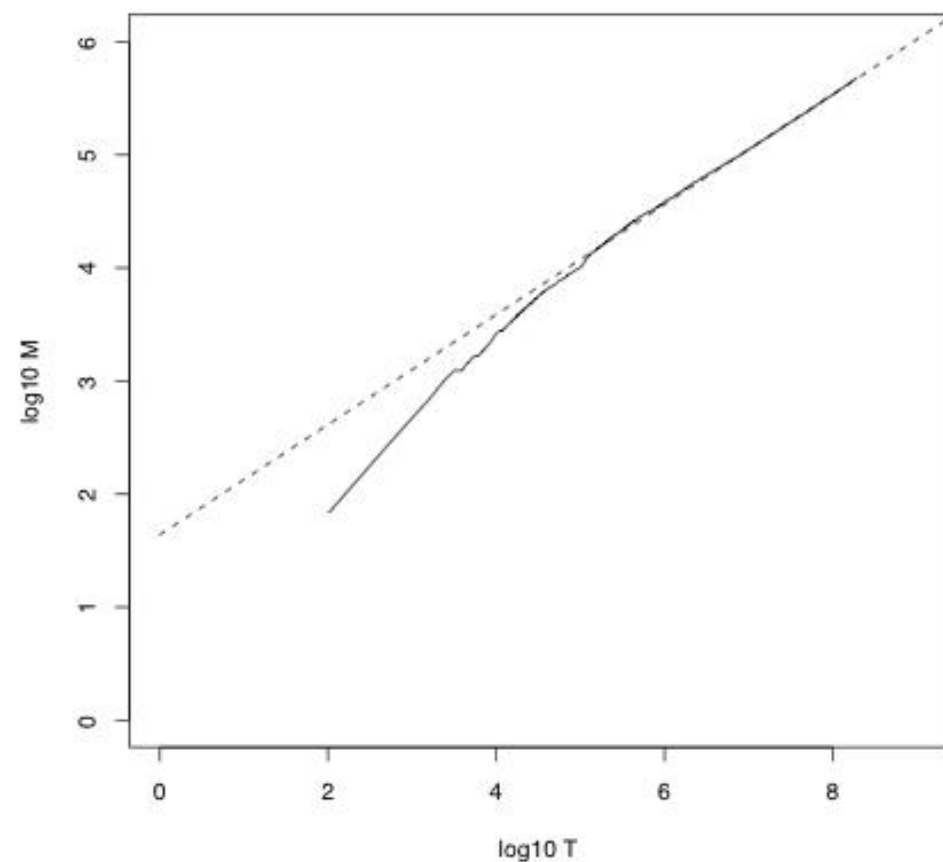
Δ %： 与上一步相比的变化百分比

T%： 与未过滤相比的变化百分比

第一个问题：词汇表有多大(即词项数目)?

- 即有多少不同的单词数目?
 - 首先，能否假设这个数目存在一个上界?
 - 不能：对于长度为20的单词，有大约 $70^{20} \approx 10^{37}$ 种可能的单词
- 实际上，词汇表大小会随着文档集的大小增长而增长！
- Heaps定律: $M = kT^b$
 - M 是词汇表大小, T 是文档集的大小(所有词条的个数，即所有文档大小之和)
- 参数 k 和 b 的一个经典取值是: $30 \leq k \leq 100$ 及 $b \approx 0.5$.
- Heaps定律在对数空间下是线性的
 - 这也是在对数空间下两者之间最简单的关系
 - 经验规律

Reuters RCV1上的Heaps定律



- 实线：真实分布；虚线：拟合分布
- 词汇表大小 M 是文档集规模 T 的一个函数
- 图中通过最小二乘法拟合出的直线方程为：

$$\log_{10} M = 0.49 * \log_{10} T + 1.64$$

- 于是有：
- $M = 10^{1.64} T^{0.49}$
- $k = 10^{1.64} \approx 44$
- $b = 0.49$

拟合 vs. 真实

- 例子: 对于前1,000,020个词条, 根据Heaps定律预计将有38,323个词项:

$$44 \times 1,000,020^{0.49} \approx 38,323$$

- 实际的词项数目为38,365, 和预测值非常接近
- 经验上的观察结果表明, 一般情况下拟合度还是非常高的

课堂练习

- ① 在容许拼写错误或者对拼写错误自动纠错的情况下，Heaps定律的效果如何？
- ② 计算词汇表大小 M
 - 观察一个网页集合，你会发现在前10000个词条中有3000个不同的词项，在前1000000个词条中有30000个不同的词项
 - 假定某搜索引擎索引了总共20,000,000,000 (2×10^{10})个网页，平均每个网页包含200个词条
 - 那么按照Heaps定律，被索引的文档集的词汇表大小是多少？

课堂练习

存在拼写错误：会增加词项数目

自动纠错：总体词项数目趋于正常

对效果有一定影响，但是除非存在大量拼写错误，否则不会有显著影响。

$$3000 = k * 10000^b;$$

$$30000 = k * 1M^b;$$

求解得到 $k=30$, $b=0.5$

$$\text{代入 } M = k * (200 * 2 * 10^{10})^b$$

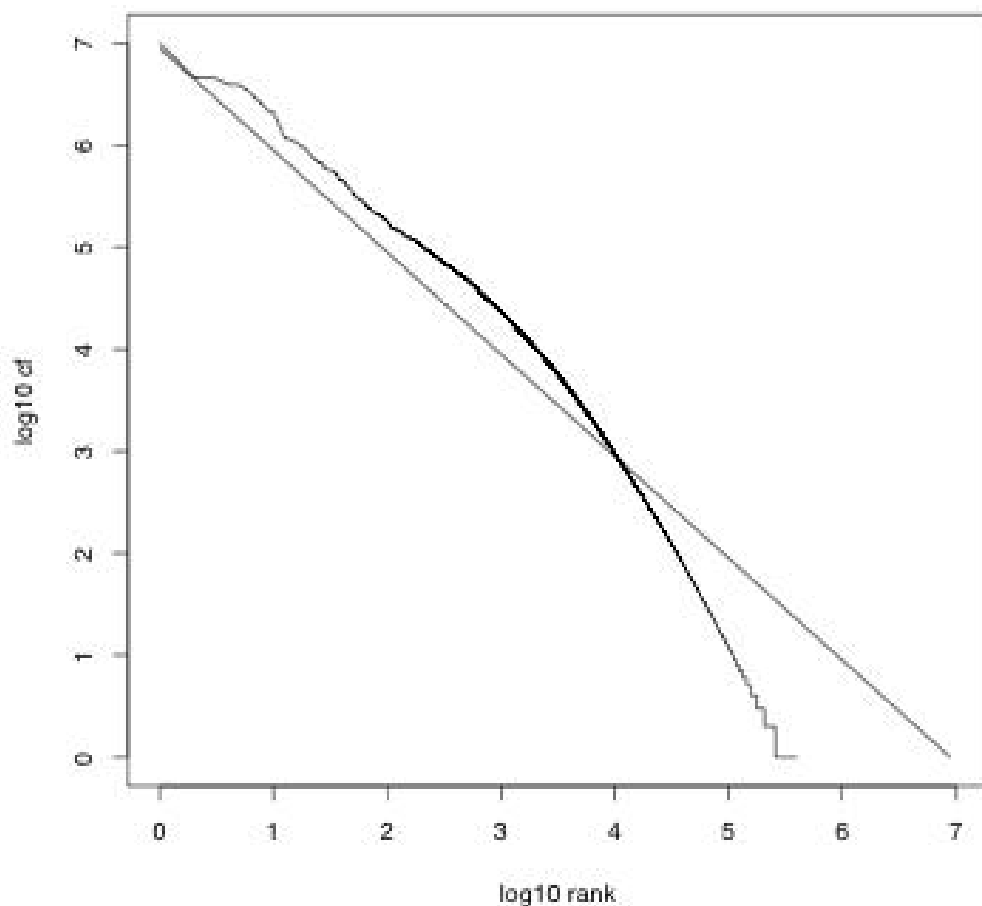
第二个问题：词项的分布如何？Zipf定律

- Heaps定律告诉我们随着文档集规模的增长词项的增长情况
- 但是我们还需要知道在文档集中有多少高频词项 vs. 低频词项。
- 在自然语言中，有一些极高频词项，有大量极低频的罕见词项

Zipf定律

- Zipf定律: 第*i*常见的词项的频率 cf_i 和 $1/i$ 成正比
- $cf_i \propto \frac{1}{i}$
- cf_i 是语料中词项频率(collection frequency): 词项 t_i 在所有文档中出现的次数(不是出现该词项的文档数目 df).
- 于是, 如果最常见的词项(*the*)出现 cf_1 次, 那么第二常见的词项 (*of*)出现次数 $cf_2 = \frac{1}{2}cf_1 \dots$
- ... 第三常见的词项 (*and*) 出现次数为 $cf_3 = \frac{1}{3}cf_1$
- 另一种表示方式: $cf_i = c * i^k$ 或 $\log cf_i = \log c + k \log i$ ($k = -1$)
- 幂定律(power law)的一个实例

Reuters RCV1上Zipf定律的体现



拟合度不是非常高，但是
最重要的是如下关键性发现：
高频词项很少，低频
罕见词项很多

提纲

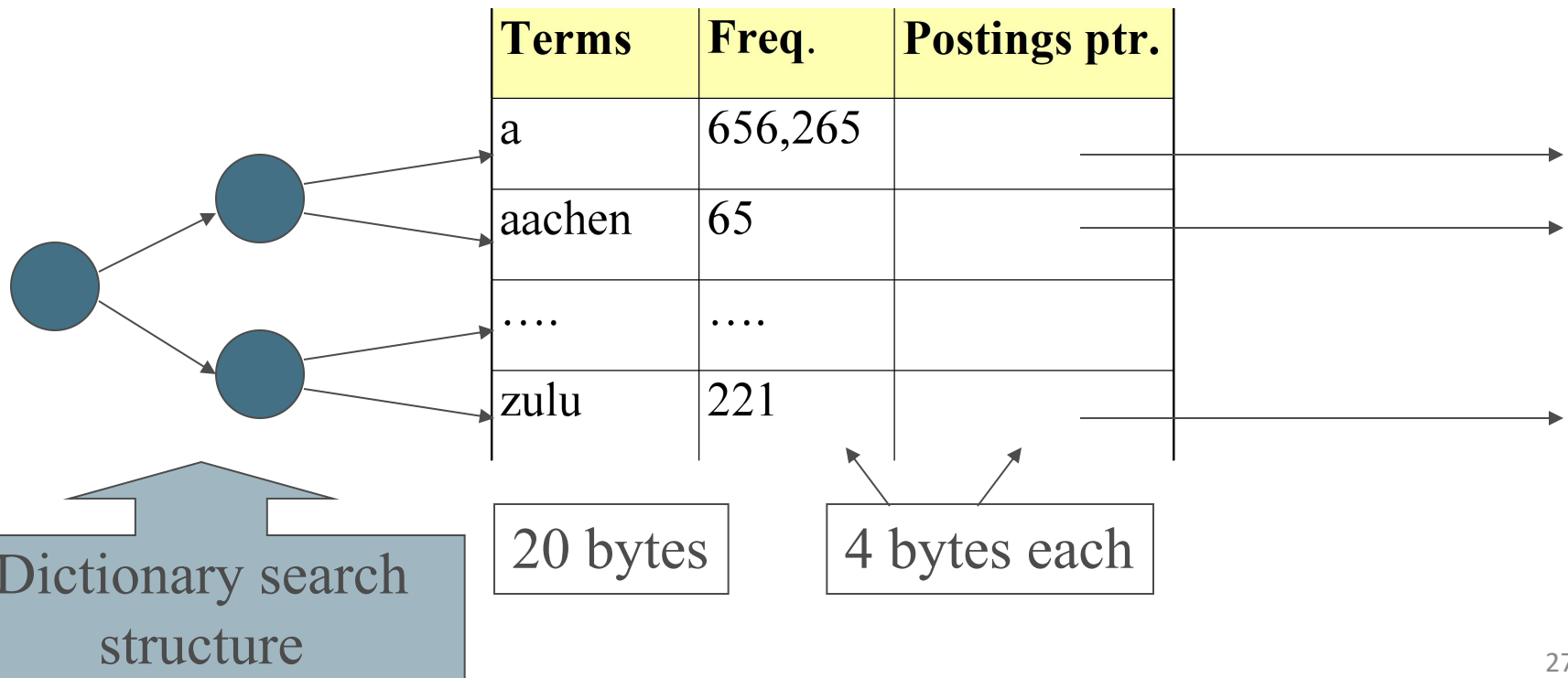
- ① 上一讲回顾
- ② 压缩
- ③ 词项统计量
- ④ 词典压缩
- ⑤ 倒排记录表压缩

词典压缩

- 一般而言，相对于倒排记录表，词典所占空间较小
- 但是我们想将词典放入内存
- 另外，满足一些特定领域特定应用的需要，如手机、机载计算机上的应用或要求快速启动等需求
- 因此，压缩词典相当重要

回顾：定长数组方式下的词典存储

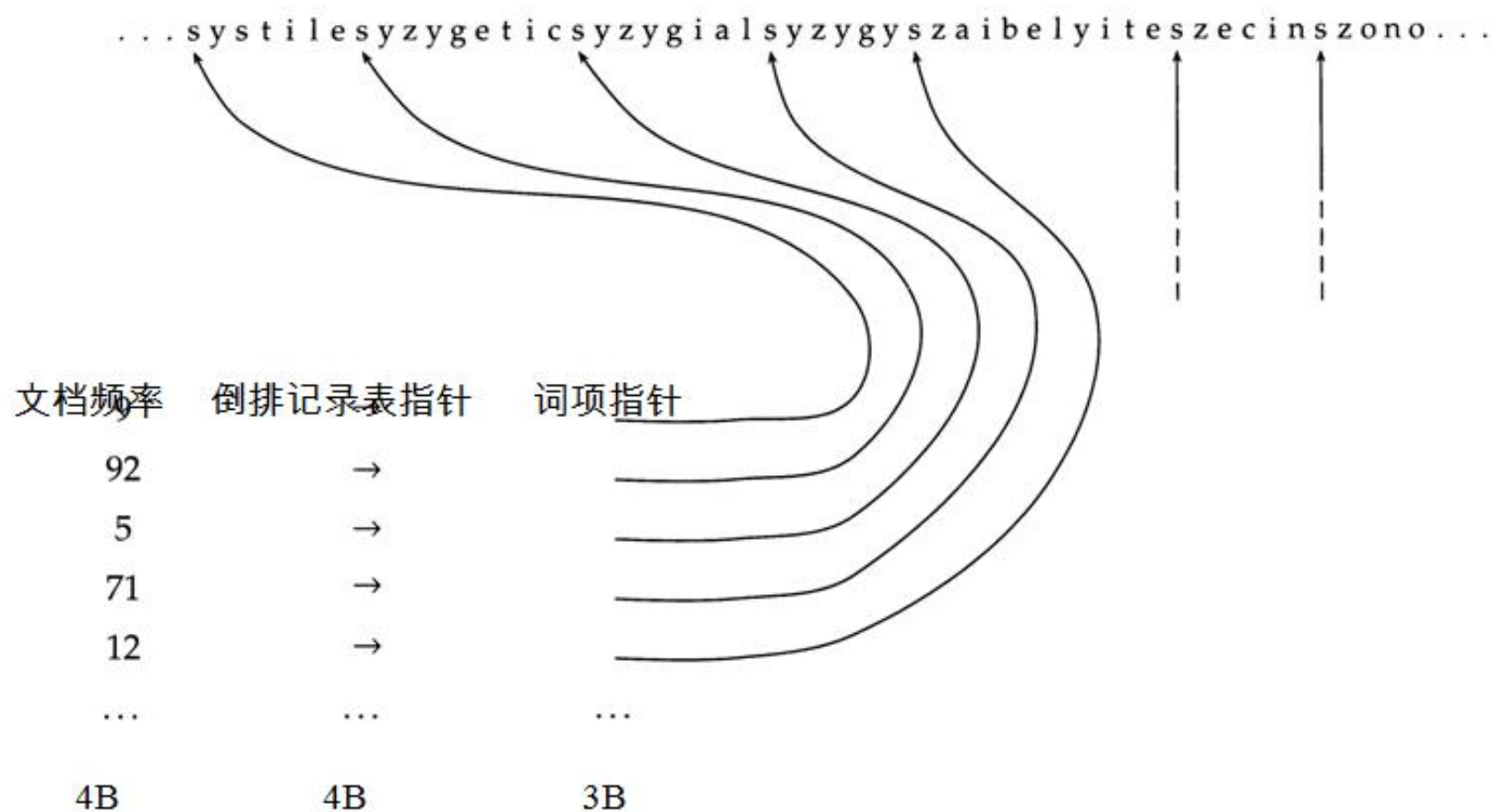
- 每个词项需要 $20+4+4=28$ 个字节
 - 对于RCV1, $\sim 400,000$ terms; $28 \text{ bytes/term} = 11.2 \text{ MB}$.
- 词项查找：典型的二叉树搜索



定长方式的不足

- 大量存储空间被浪费
 - 即使是长度为1的词条，我们也分配20个字节
- 不能处理长度大于20字节的词条，如
HYDROCHLOROFLUOROCARBONS 和
SUPERCALIFRAGILISTICEXPIALIDOCIOUS
- 而英语中每个词条的平均长度为8个字符
- 能否对每个词条平均只使用8个字节来存储？

将整部词典看成单一字符串(Dictionary as a string)



单一字符串方式下的空间消耗

- 每个词项的词项频率需要4个字节
- 每个词项指向倒排记录表的指针需要4个字节
- 每个词项平均需要8个字节
- 指向字符串的指针需要3个字节 (8×400000 个位置需要 $\log_2 (8 * 400000) < 24$ 位来表示)
- 空间消耗: $400,000 \times (4 + 4 + 3 + 8) = 7.6\text{MB}$ (而定长数组方式需要11.2MB)

单一字符串方式下按块存储

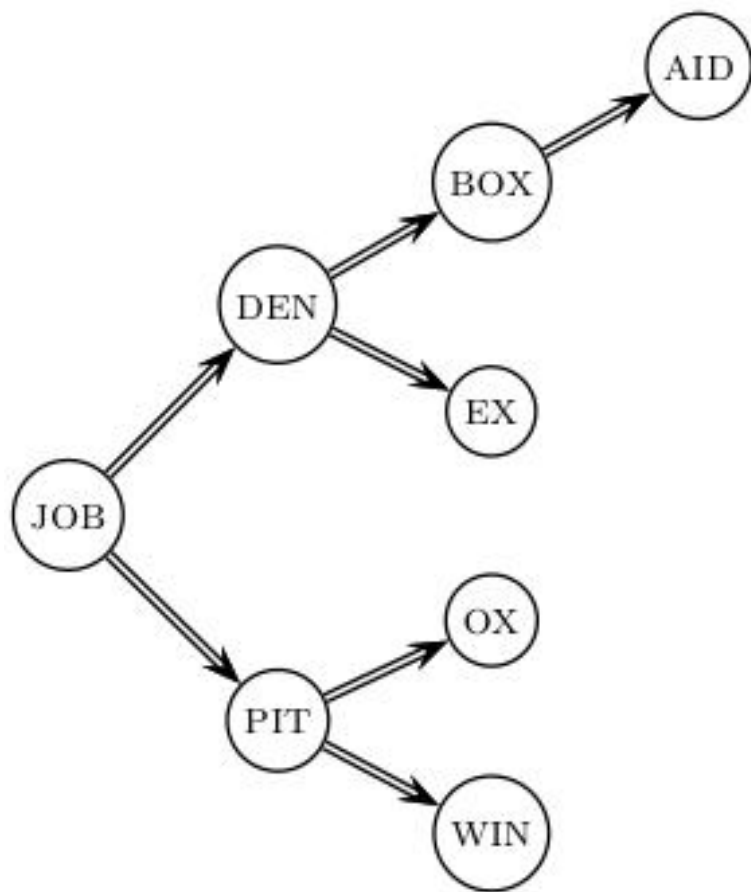
...7systile9syzygetic8syzygial6syzygy11szaibelyite6szecin...

文档频率	倒排记录表指针	词项指针
9	→	
92	→	
5	→	
71	→	
12	→	
...

按块存储下的空间消耗

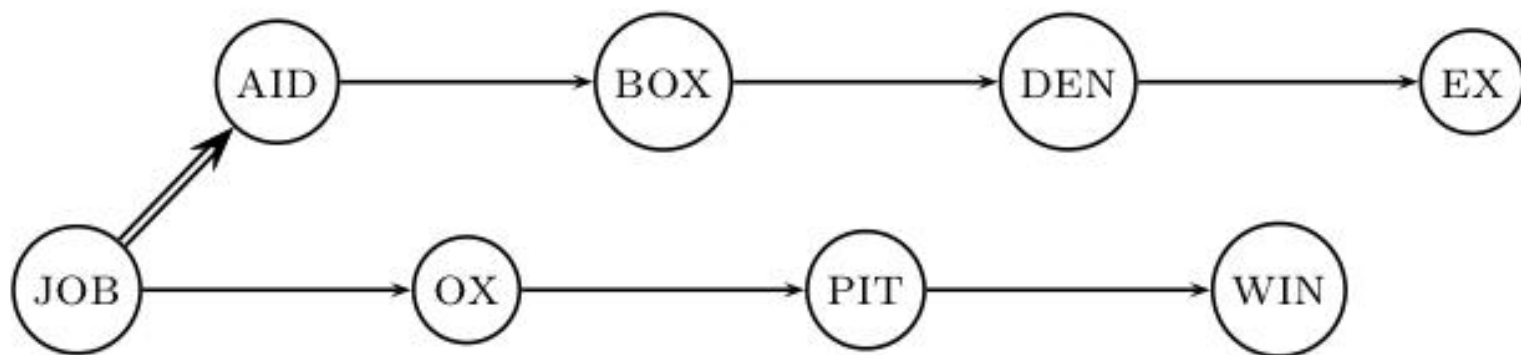
- 如果不按块存储，则每4个词项指针将占据空间 $4 \times 3 = 12\text{B}$
- 现在按块存储，假设块大小 $k=4$ ，此时每4个词项只需要保留1个词项指针，但是同时需要增加4个字节来表示每个词项的长度，此时每4个词项需要 $3+4=7\text{B}$
- 因此，每4个词项将节省 $12-7=5\text{B}$
- 于是，整个词典空间将节省 $400,000/4 \times 5\text{B} = 0.5\text{MB}$
- 最终的词典空间将从 7.6MB 压缩至 7.1MB

不采用块存储方式下的词项查找



典型的二叉查找

采用块存储方式下的词项查找：稍慢



因此块不能太大，否则影响搜索效率

前端编码(Front coding)

- 每个块当中 ($k=4$), 会有公共前缀 ...
- 8 a u t o m a t a 8 a u t o m a t e 9 a u t o m a t i c 10 a u t o m a t i o n
- \Downarrow
- ... 可以采用前端编码方式继续压缩
- $8 a u t o m a t * a 1 \diamond e 2 \diamond i c 3 \diamond i o n$
- 上面前端编码中, 第一个数字8表示第一个词项长度, 后面的数字1、2、3分别表示 (除公共前缀外) 词项剩余部分长度。星号与菱形是特殊分隔符

Reuters RCV1词典压缩情况总表

数据结构	压缩后的空间大小（单位：MB）
词典，定长数组	11.2
词典，长字符串+词项指针	7.6
词典，按块存储， $k=4$	7.1
词典，按块存储+前端编码	5.9

课堂练习

- 哪些前缀应该用于前端编码？需要在哪些方面有所权衡？
 - 输入：词项列表，即词汇表
 - 输出：用于前端编码的前缀列表

提纲

- ① 上一讲回顾
- ② 压缩
- ③ 词项统计量
- ④ 词典压缩
- ⑤ 倒排记录表压缩

倒排记录表压缩

- 倒排记录表空间远大于词典，至少10倍以上
- 压缩关键：对每条倒排记录进行压缩
- 目前每条倒排记录表中存放的是docID.
- 对于Reuters RCV1(800,000篇文档), 当每个docID可以采用4字节（即32位）整数来表示
- 当然，我们也可以采用 $\log_2 800,000 \approx 19.6 < 20$ 位来表示每个docID.
- 我们的压缩目标是：压缩后每个docID用到的位数远小于20比特

倒排记录的两个极端情况

- 类似 *arachnocratic* 这样的词仅在百万分之一的文档中出现，我们使用 $\log_2 1M \approx 20$ bits 进行存储
- 而类似 *the* 这样的高频词几乎在所有文档中出现，因此使用 $20 \text{ bits/posting} \approx 2\text{MB}$ 存储的代价昂贵。
 - 这种情况更适合使用 0/1 位向量(bitmap vector) ($\approx 100\text{K}$)

关键思想: 存储docID间隔而不是docID本身

- 每个倒排记录表中的docID是从低到高排序
 - 例子: COMPUTER: 283154, 283159, 283202, ...
- 存储间隔能够降低开销: $283159 - 283154 = 5$, $283202 - 283154 = 43$
- 于是可以顺序存储间隔(第一个不是间隔): COMPUTER: 283154, 5, 43, ...
- 高频词项的间隔较小
- 因此, 可以对这些间隔采用小于20比特的存储方式

对间隔编码

	编码对象	倒排记录表				
the	文档ID	...	283 042	283 043	283 044	283 045 ...
	文档ID间距		1	1	1	...
computer	文档ID	...	283 047	283 154	283 159	283 202 ...
	文档ID间距		107	5	43	...
arachnogenic	文档ID	252 000	500 100			
	文档ID间距	252 000	248 100			

变长编码

- 目标:
 - 对于 ARACHNOCENTRIC 及其他罕见词项, 对每个间隔仍然使用20比特 (bit位)
 - 对于THE及其他高频词项, 每个间隔仅仅使用很少的比特位来编码
- 为了实现上述目标, 需要设计一个变长编码(variable length encoding)
- 可变长编码对于小间隔采用短编码而对于长间隔采用长编码

可变字节(VB)码

- 被很多商用/研究系统所采用
- 变长编码及对齐敏感性(指匹配时按字节对齐还是按照位对齐)的简单且不错的混合产物
- 设定一个专用位 (高位) c 作为延续位(continuation bit)
- 如果间隔表示少于7比特, 那么 c 置 1, 将间隔编入一个字节的后7位中
- 否则: 将高7位放入当前字节中, 并将 c 置 0, 剩下的位数采用同样的方法进行处理, 最后一个字节的 c 置1 (表示结束)
 - 从高到低编码

VB 编码的例子

文档ID	824	829	215 406
间距		5	214 577
VB编码	00000110 10111000	10000101	00001101 00001100 10110001

VB 编码算法

VBENCODENUMBER(n)

```

1   $bytes \leftarrow \langle \rangle$ 
2  while  $true$ 
3  do PREPEND( $bytes, n \bmod 128$ )
4      if  $n < 128$ 
5          then BREAK
6       $n \leftarrow n \text{ div } 128$ 
7   $bytes[\text{LENGTH}(bytes)] += 128$ 
8  return  $bytes$ 

```

VBENCODE($numbers$)

```

1   $bytestream \leftarrow \langle \rangle$ 
2  for each  $n \in numbers$ 
3  do  $bytes \leftarrow \text{VBENCODENUMBER}(n)$ 
4       $bytestream \leftarrow \text{EXTEND}(bytestream, bytes)$ 
5  return  $bytestream$ 

```

$130 \bmod 128 = 2 \rightarrow bytes$ 数组

$130 \text{ div } 128 = 1$, prepend 到 $bytes$ 数组

于是循环结束有 $bytes = [1, 2]$

算法最后一步，是在 $bytes[\text{length}(bytes)]$ 上加128，即延续位置1

VB编码的解码算法

VBDECODE(*bytestream*)

```
1  numbers  $\leftarrow \langle \rangle$ 
2  n  $\leftarrow 0$ 
3  for i  $\leftarrow 1$  to LENGTH(bytestream)
4  do if bytestream[i] < 128
5      then n  $\leftarrow 128 \times n + \text{bytestream}[i]$ 
6      else n  $\leftarrow 128 \times n + (\text{bytestream}[i] - 128)$ 
7          APPEND(numbers, n)
8          n  $\leftarrow 0$ 
9  return numbers
```

当延续位为1, $\text{bytestream}[i] > 128$, 因此 if $\text{bytestream}[i] < 128$ 判断的是数字之间界线 (即对齐)

其它编码

- 除字节外，还可以采用不同的对齐单位：比如32位 (word)、16位及4位 (nibble) 等等
- 如果有很多很小的间隔，那么采用可变字节码会浪费很多空间，而此时采用4位为单位将会节省空间
- 最近一些工作采用了32位的方式 - 参考讲义末尾的参考材料

γ 编码

- [illegible]

[illegible]

γ 编码

- 将G (Gap, 间隔) 表示成长度(length)和偏移(offset)两部分
- 偏移对应G的二进制编码，只不过将首部的1去掉
- 例如 $13 \rightarrow 1101 \rightarrow 101 = \text{偏移}$
- 长度部分给出的是偏移的位数
- 比如G=13 (偏移为 101), 长度部分位数为 3
- 长度部分采用一元编码: 1110.
- 于是G的 γ 编码就是将长度部分和偏移部分两者联接起来得到的结果。

γ 编码的例子

数 字	一元编码	长 度	偏 移	γ 编 码
0	0			
1	10	0		0
2	110	10	0	10,0
3	1110	10	1	10,1
4	11110	110	00	110,00
9	111111110	1110	001	1110,001
13		1110	101	1110,101
24		11110	1000	11110,1000
511		111111110	11111111	111111110,11111111
1025		11111111110	0000000001	11111111110 ,0000000001

课堂练习

- 计算130的可变字节码
- 计算130的 r 编码

130二进制表示: 10000010

偏移: 0000010 长度部分: 11111110

最终编码: 11111110 0000010

VB编码: 00000001 10000010

Υ编码的长度

- 偏移部分是 $\log_2 G$ 比特位
- 长度部分需要 $\log_2 G + 1$ 比特位
- 因此，全部编码需要 $2\log_2 G + 1$ 比特位
- Υ 编码的长度均是奇数
- Υ 编码在最优编码长度的2倍左右
 - 假定间隔G的出现频率正比于 $\log_2 G$ ——实际中并非如此)
 - (assuming the frequency of a gap G is proportional to $\log_2 G$ – not really true)

γ 编码的性质

- γ 编码是前缀无关的，也就是说一个合法的 γ 编码不会 是任何一个其他的合法 γ 编码的前缀，也保证了解码的 唯一性。
- 编码在最优编码的2或3倍之内
 - 上述结果并不依赖于间隔的分布！
- 因此， γ 编码适用于任何分布，也就说 γ 编码是通用性 (universal) 编码
- γ 编码是无参数编码，不需要通过拟合得到参数

γ 编码的对齐问题

- 机器通常有字边界 – 8, 16, 32 位
- 按照位进行压缩或其他处理可能会较慢
- 可变字节码通常按字边界对齐，因此可能效率更高
- 除去效率高之外，可变字节码虽然额外增加了一点点开销，但是在概念上也要简单很多
- 因此在商用系统中VB编码更常见

Reuters RCV1索引压缩总表

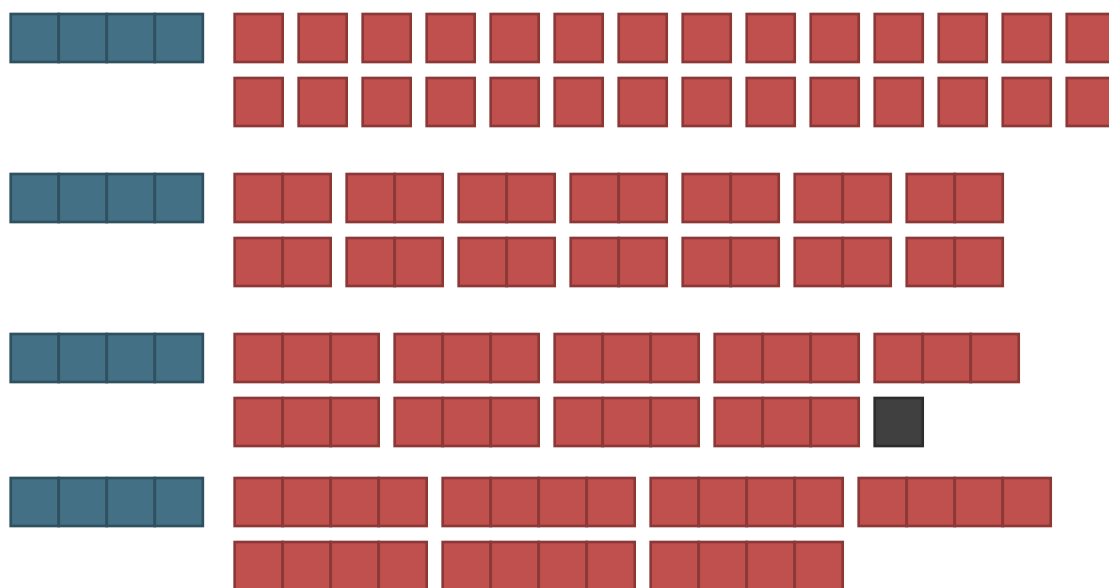
数据结构	压缩后的空间大小（单位：MB）
词典，定长数组	11.2
词典，长字符串+词项指针	7.6
词典，按块存储， $k=4$	7.1
词典，按块存储+前端编码	5.9
文档集（文本、XML标签等）	3 600.0
文档集（文本）	960.0
词项关联矩阵	40 000.0
倒排记录表，未压缩（32位字）	400.0
倒排记录表，未压缩（20位）	250.0
倒排记录表，可变字节码	116.0
倒排记录表， γ 编码	101.0

Group Variable Integer code (组变长整数编码)

- Google 在2000年左右使用的算法
 - Jeff Dean, keynote at WSDM 2009 and presentations at CS276
- 按块存储。每块大小为5-17个字节，存放4个整数编码
- 每块首字节: 4个2位的二进制长度
 $L_1 | L_2 | L_3 | L_4$, $L_j \in \{1, 2, 3, 4\}$
- 接下来, 使用 $L_1 + L_2 + L_3 + L_4$ bytes (between 4–16) 存放4个整数
 - 每个整数占用 8/16/24/32 位，取决于整数的实际大小. 可存储的最大间隔约为 ~4 billion
- 据称比VB编码快两倍
 - 间隔解码更简单 – 这是因为没有位masking, 无延续位
 - 首字节可使用查找表(lookup table)或switch解码

Simple-9 [Anh & Moffat, 2004]

怎样在32位中存放多个数字，How can we store several numbers in 32 bits with a format selector?



Simple9 Encoding Scheme [Anh & Moffat, 2004]

- 编码块: 4 字节(32 位), 通常一个word的占用空间
- “最显著点”(most significant nibble, 即前4 位) 概括了剩余28 位的结构:

- 0: 1个 28位数字
- 1: 2个 14位 数字
- 2: 3个 9位 数字 (1个空位)
- 3: 4个 7位 数字
- 4: 5个 5位 数字 (3个空位)
- 5: 7个 4位 数字
- 6: 9个 3位 数字 (1个空位)
- 7: 14个 2位 数字
- 8: 28个 1位 数字

Layout (4 bits)	n numbers of b bits each $n * b \leq 28$
--------------------	---

每块4字节, 前4位标识块内结构, 剩余28位存储若干个数字, 每个数字占用相同的位数

最大可存储 2^{28}

- Simple16 是Simple9的改进: 多出了 5 个非平均结构配置
- 解码效率高
- 可扩展性强 – 例如可以扩展到64位编码

总结

- 现在我们可以构建一个空间上非常节省的支持高效布尔检索的索引
- 大小仅为文档集中文本量的10-15%
- 然而，这里我们没有考虑词项的出现位置和频率信息
- 因此，实际当中并不能达到如此高的压缩比

参考资料

- 《信息检索导论》 第5章
- <http://ifnlp.org/ir>
 - 有关字对齐二元编码的原文Anh and Moffat (2005); 及 Anh and Moffat (2006a)
 - 有关可变字节码的原文Scholer, Williams, Yiannis and Zobel (2002)
 - 更多的有关压缩 (包括位置和频率信息的压缩)的细节参考 Zobel and Moffat (2006)