第4讲 索引构建

Index construction

*节选自王斌老师《现代信息检索》课程的课件

提纲

1. 上一讲回顾
2. 简介
3. BSBI算法
4. SPIMI算法
5. 分布式索引构建
6. 动态索引构建

提纲

1. 上一讲回顾
2. 简介
3. BSBI算法
4. SPIMI算法
5. 分布式索引构建
6. 动态索引构建

上一讲内容

- 词典的数据结构：
  - 哈希表 vs. 树结构
- 容错式检索(Tolerant retrieval):
  - 通配查询：包含通配符*的查询
    - 轮排索引 vs. k-gram索引
  - 拼写校正：
    - 编辑距离 vs. k-gram相似度
    - 词独立校正法 vs. 上下文敏感校正法
    - Soundex算法

采用定长数组法存储词典

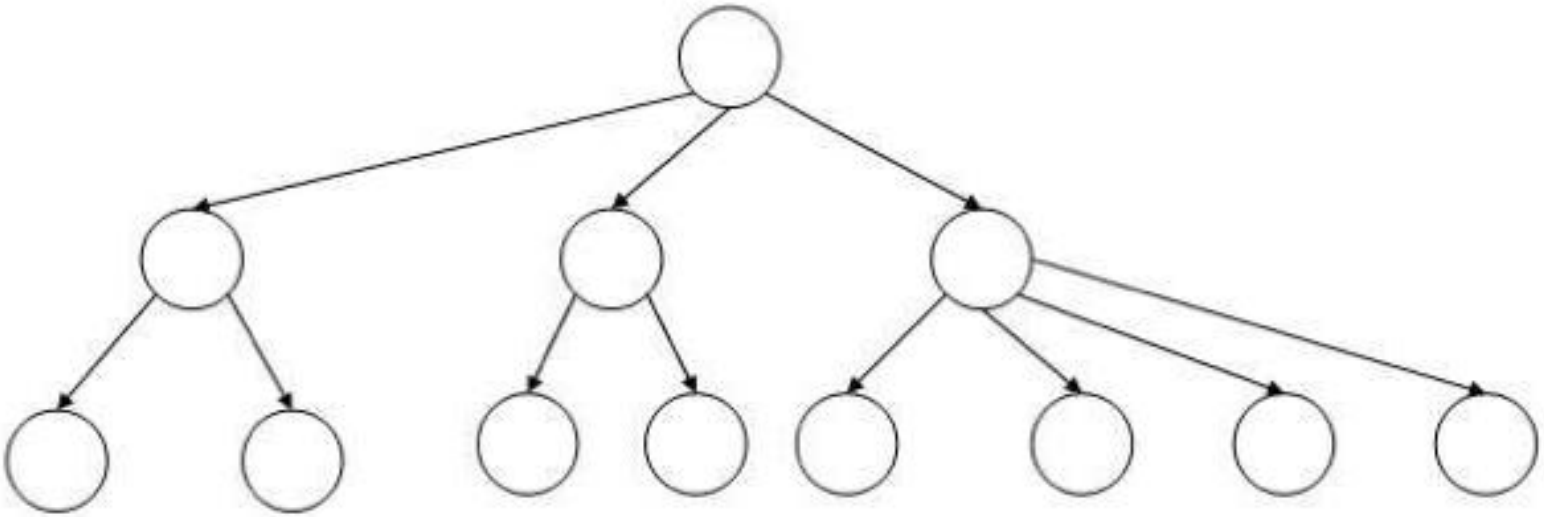
term	document frequency	pointer to postings list
a	656,265	⟶
aachen	65	⟶
. . .	. . .	. . .
zulu	221	⟶

空间消耗：　20字节　4字节　4字节

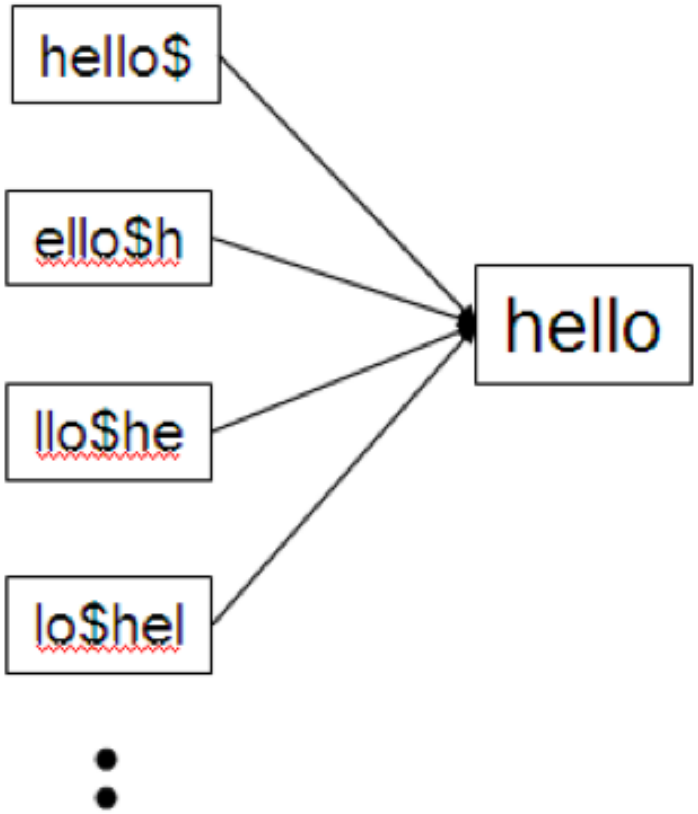
支持词典查找的两种数据结构

- 哈希表：
  - 定位速度快，常数时间
  - 不宜支持动态变化的词典
  - 不支持前缀查询
- 树结构：二叉树、B-树等等
  - 定位速度为指数时间
  - 二叉(平衡)树支持动态变化，但是重排代价大。B-树能否缓解上述问题
  - 支持前缀查询

基于B-树的词典查找



基于轮排索引的通配查询处理



查询:

- 对 X, 查找 X$
- 对X*, 查找 X*$
- 对*X, 查找 X$*
- 对*X*, 查找 X*
- 对 X*Y, 查找 Y$X*

基于k-gram索引的通配查询处理

- 比轮排索引空间开销要小
- 枚举一个词项中所有连读的k个字符构成的k-gram 。
- 2-gram称为二元组(bigram)
- 例子: from April is the cruelest month we get the bigrams: $a ap pr ri il l$ $i is s$ $t th he e$ $c cr ru ue el le es st t$ $m mo on nt h$
- 同前面一样，$ 是一个特殊字符
- 构建一个倒排索引，此时词典部分是所有的2-gram，倒排记录表部分是包含某个2-gram的所有词项
- 相当于对词项再构建一个倒排索引(二级索引)

Levenshtein 距离计算

LevenshteinDistance(s_1, s_2)

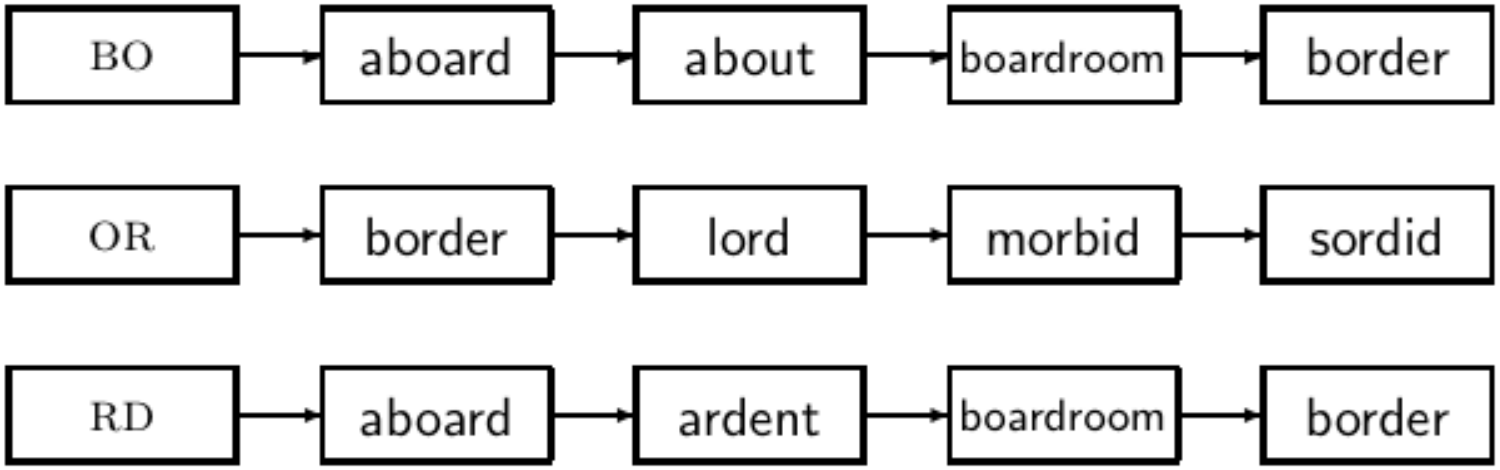
```
 1  for i ← 0 to |s1|
 2  do m[i, 0] = i
 3  for j ← 0 to |s2|
 4  do m[0, j] = j
 5  for i ← 1 to |s1|
 6  do for j ← 1 to |s2|
 7       do if s1[i] = s2[j]
 8            then m[i, j] = min{m[i − 1, j] + 1, m[i, j − 1] + 1, m[i − 1, j − 1]}
 9            else  m[i, j] = min{m[i − 1, j] + 1, m[i, j − 1] + 1, m[i − 1, j − 1] + 1}
10  return m[|s1|, |s2|]
```

Operations: insert, delete, replace, copy

基于编辑距离的拼写校正

- 给定查询词，穷举词汇表中和该查询的编辑距离(或带权重的编辑聚类)低于某个预定值的所有单词
- 求上述结果和给定的某个“正确”词表之间的交集
- 将交集结果推荐给用户
- 代价很大，实际当中往往通过启发式策略提高查找效率(如：保证两者之间具有较长公共子串)

基于*k*-gram索引的拼写校正： *bordroom*



本讲内容

- 两种索引构建算法: BSBI (简单) 和 SPIMI (更符合实际情况)
- 分布式索引构建: MapReduce
- 动态索引构建: 如何随着文档集变化更新索引

提纲

硬件基础知识

- 信息检索系统中的很多设计上的决策取决于硬件限制
- 首先简单介绍本课程中需要用到的硬件知识

硬件基础知识

- 在内存中访问数据会比从硬盘访问数据快很多(大概10倍左右的差距)
- 硬盘寻道时间是闲置时间：磁头在定位时不发生数据传输
- 为优化从磁盘到内存的传送时间，一个大(连续)块的传输会比多个小块(非连续)的传输速度快
- 硬盘 I/O是基于块的: 读写时是整块进行的。块大小：8KB到256 KB不等
- IR系统的服务器的典型配置是几个GB的内存，有时内存可能达到几十GB，数百G或者上T的硬盘。
- 容错处理的代价非常昂贵：采用多台普通机器会比一台提供容错的机器的价格更便宜

Reuters RCV1 语料库

- 《莎士比亚全集》规模较小，用来构建索引不能说明问题
- 本讲使用Reuters RCV1文档集来介绍可扩展的索引构建技术
- 路透社 1995到1996年一年的英语新闻报道

一篇Reuters RCV1文档的样例

REUTERS

You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly En

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprir

[-] Text [-

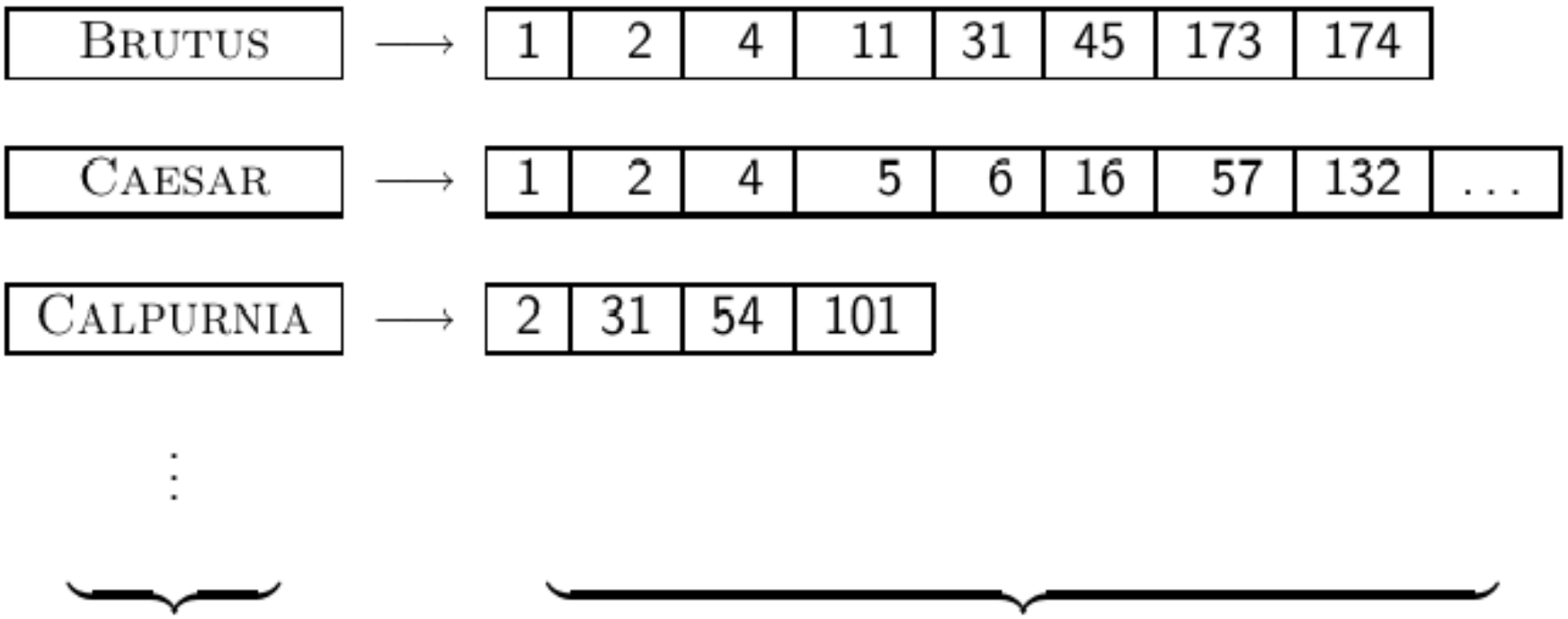
SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian

提纲

1. 上一讲回顾
2. 简介
3. BSBI算法
4. SPIMI算法
5. 分布式索引构建
6. 动态索引构建

目标: 构建倒排索引


BRUTUS
1 2 4 11 31 45 173 174
CAESAR
1 2 4 5 6 16 57 132 . . .
CALPURNIA
2 31 54 101
词典
倒排记录表

第一讲中介绍的索引构建: 在内存中对倒排记录表进行排序(基于排序的索引构建方法)

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

基于排序的索引构建方法

- 在构建索引时，每次分析一篇文档
- 对于每个词项而言，其倒排记录表不到最后一篇文档都是不完整的。
- 那么能否在最后排序之前将前面产生的倒排记录表全部放在内存中？
- 答案显然是否定的，特别是对大规模的文档集来说
- 如果每条倒排记录占10–12个字节，那么对于大规模语料，需要更大的存储空间
- 以RCV1为例，$T = 100,000,000$，这些倒排记录表倒是可以放在2010年的一台典型配置的计算机的内存中
- 但是这种基于内存的索引构建方法显然无法扩展到大规模文档集上
- 因此，需要在磁盘上存储中间结果

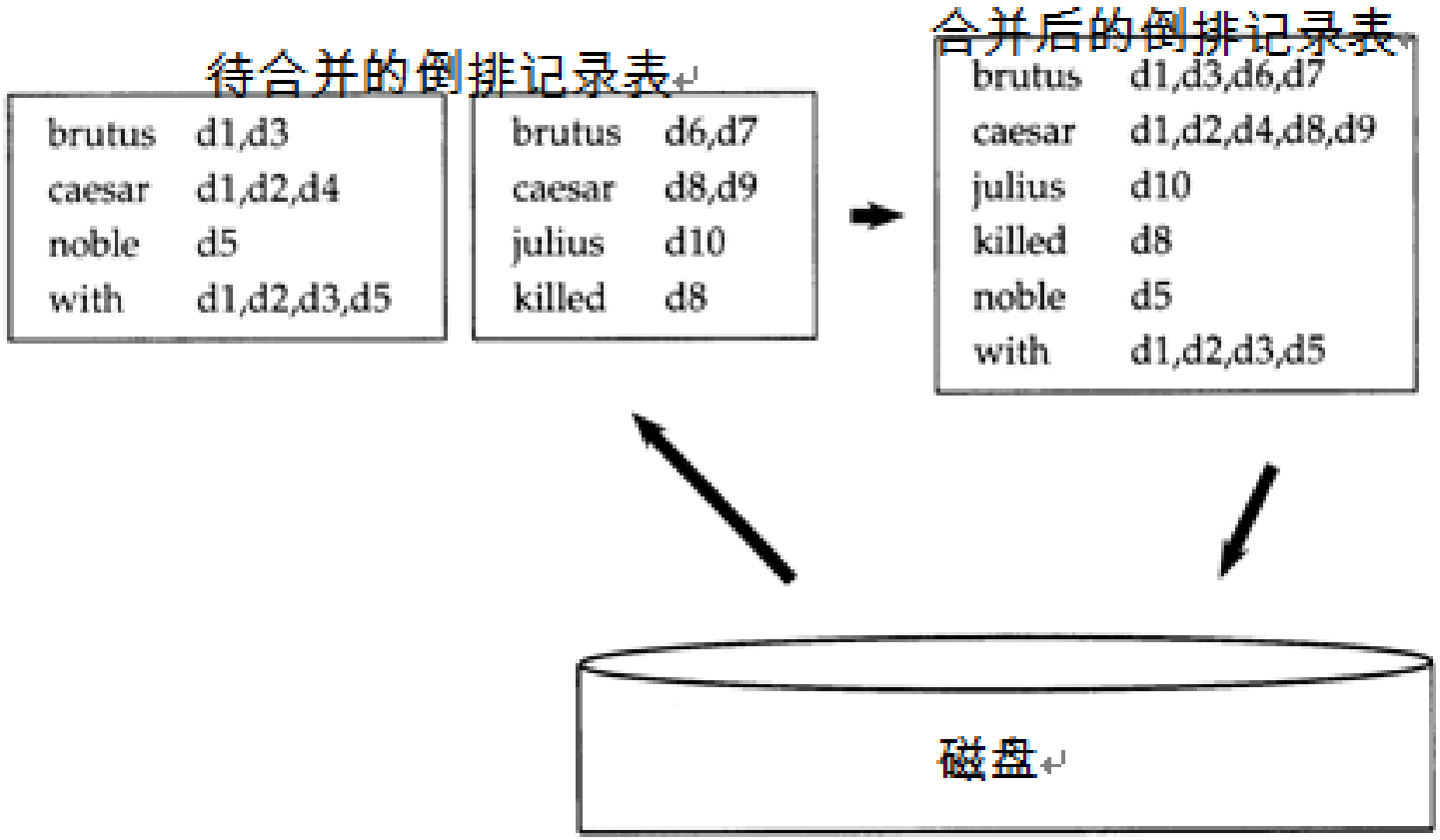
是否在磁盘上采用同样的算法？

- 能否使用前面同样的算法，但是是在磁盘而不是内存中完成排序?
- 不可能，这是因为对 $T = 100,000,000$条记录在磁盘上进行那个排序需要太多的磁盘寻道过程.
- 需要一个外部排序算法

外部排序算法中磁盘寻道次数很少

- 需要对T = 100,000,000条无位置信息的倒排记录进行排序
  - 每条倒排记录需要12字节 (4+4+4: termID, docID, df)
- 定义一个能够包含10,000,000条上述倒排记录的数据块
  - 这个数据块很容易放入内存中(12*10M=120M)
  - 对于RCV1有10个数据块
- 算法的基本思路:
  - 对每个块: (i) 倒排记录累积到10,000,000条, (ii) 在内存中排序, (iii) 写回磁盘
  - 最后将所有的块合并成一个大的有序的倒排索引

两个块的合并过程


待合并的倒排记录表
brutus d1,d3
caesar d1,d2,d4
noble d5
with d1,d2,d3,d5
brutus d6,d7
caesar d8,d9
julius d10
killed d8
合并后的倒排记录表
brutus d1,d3,d6,d7
caesar d1,d2,d4,d8,d9
julius d10
killed d8
noble d5
with d1,d2,d3,d5
磁盘

基于块的排序索引构建算法BSBI（Blocked Sort-Based Indexing）

```
BSBIndexConstruction()
1  n ← 0
2  while (all documents have not been processed)
3  do n ← n + 1
4       block ← ParseNextBlock()
5       BSBI-Invert(block)
6       WriteBlockToDisk(block, f_n)
7  MergeBlocks(f_1, ..., f_n; f_merged)
```

- 该算法中有一个关键决策就是确定块的大小

提纲

基于排序的索引构建算法的问题

- 假定词典可以在内存中放下
- 通常需要一部词典(动态增长)来将term映射成termID
- 实际上，倒排记录表可以直接采用 term,docID 方式而不是termID,docID方式. . .
- . . . 但是此时中间文件将会变得很大

内存式单遍扫描索引构建算法ＳＰＩＭＩ Single-pass in-memory indexing

- 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
```

Merging of blocks is analogous to BSBI.

SPIMI: 压缩

- 如果使用压缩，SPIMI将更加高效
  - 词项的压缩
  - 倒排记录表的压缩
  - 参见下一讲

提纲

1. 上一讲回顾
2. 简介
3. BSBI算法
4. SPIMI算法
5. **分布式索引构建**
6. 动态索引构建

分布式索引构建

- 对于Web数据级别的数据建立索引 (don’t try this at home!): 必须使用分布式计算机集群
- 单台机器都是有可能出现故障的
  - 可能突然慢下来或者失效，不可事先预知
- 如何使用一批机器?

Google 数据中心(2007 estimates; Gartner)

- Google数据中心主要都是普通机器
- 数据中心均采用分布式架构，在世界各地分布
- 100万台服务器，300个处理器/核
- Google每15分钟装入 100,000个服务器.
- 每年的支出大概是每年2-2.5亿美元
- 这可能是世界上计算能力的10%!
- 在一个1000个节点组成的无容错系统中，每个节点的正常运行概率为99.9%，那么整个系统的正常运行概率是多少？
- 答案: 63%
- 假定一台服务器3年后会失效，那么对于100万台服务器，机器失效的平均间隔大概是多少？
- 答案: 不到2分钟

分布式索引构建

- 维持一台主机(Master)来指挥索引构建任务-这台主机被认为是安全的
- 将索引划分成多组并行任务
- 主机将把每个任务分配给某个缓冲池中的空闲机器来执行

并行任务

- 两类并行任务分配给两类机器：
  - 分析器(Parser)
  - 倒排器(Inverter)
- 将输入的文档集分片(split) (对应于BSBI/SPIMI算法中的块)
- 每个数据片都是一个文档子集

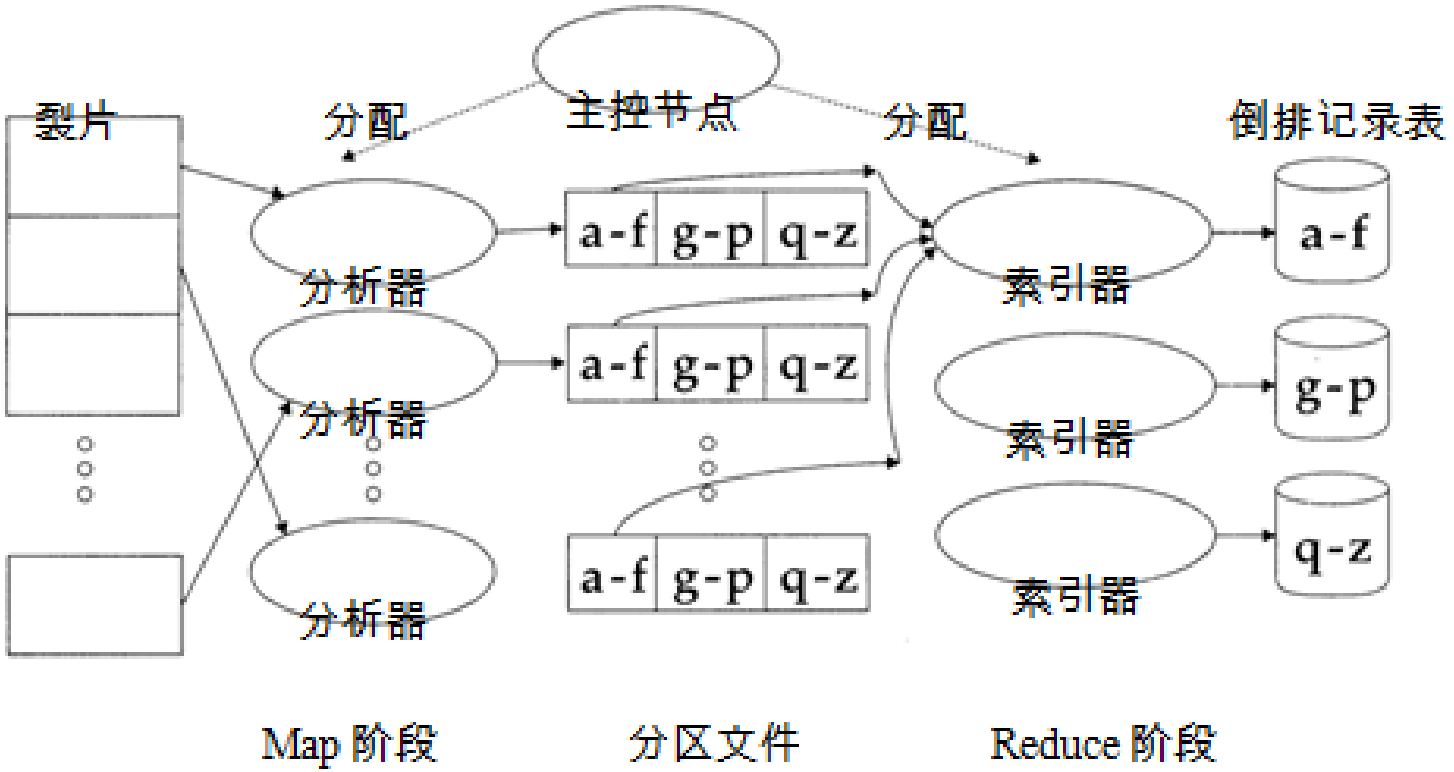
分析器（Parser）

- 主节点将一个数据片分配给一台空闲的分析器
- 分析器一次读一篇文档然后输出 (term,docID)-对
- 分析器将这些对又分成j 个词项分区
- 每个分区按照词项首字母进行划分
  - E.g., a-f, g-p, q-z (这里 j = 3)

倒排器(Inverter)

- 倒排器收集对应某一term分区(e.g., a-f分区)所有的(term,docID) 对 (即倒排记录表)
- 排序并写进倒排记录表

数据流


裂片
分配
主控节点
分配
倒排记录表
分析器
分析器
分析器
a-f
g-p
q-z
a-f
g-p
q-z
a-f
g-p
q-z
索引器
索引器
索引器
a-f
g-p
q-z
Map 阶段
分区文件
Reduce 阶段

MapReduce

- 刚才介绍的索引构建过程实际上是MapReduce的一个实例
- MapReduce是一个鲁棒的简单分布式计算框架
- Google索引构建系统 (ca. 2002) 由多个步骤组成，每个步骤都采用 MapReduce实现
- 索引构建只是一个步骤
- 另一个步骤：将按词项分割索引转换成按文档分割的索引

基于MapReduce的索引构建

Map和Reduce函数的构架	
Map: 输入	→ list(k, v)
Reduce: (k,list(v))	→ 输出
索引构建中上述构架的实例化	
Map: Web文档集	→ list(词项ID, 文档ID)
Reduce: (〈文档ID$_1$, list(docID)〉, 〈文档ID$_2$, list(docID)〉, …)	→ (倒排记录表1, 倒排记录表2,…)
索引构建的一个例子	
Map: d_2: C died. d_1: C came, C c'ed.	→ (〈C,d_2〉 〈died,d_2〉,〈C,d_1〉 〈came,d_1〉 〈C,d_1〉,〈c'ed,d_1〉)
Reduce: (〈C,(d_2,d_1,d_1)〉,〈died, (d_2)〉,〈came, (d_1)〉,〈c'ed,(d_1)〉)	→ (〈C,(d_1:2,d_2:1)〉,〈died, (d_2:1)〉,〈came,(d_1:1)〉, 〈c'ed,(d_1:1)〉)

提纲

动态索引构建

- 到目前为止，我们都假定文档集是静态的。
- 实际中假设很少成立：文档会增加、删除和修改。
- 这也意味着词典和倒排记录表必须要动态更新。

动态索引构建: 最简单的方法

- 在磁盘上维护一个大的主索引(Main index)
- 新文档放入内存中较小的辅助索引（Auxiliary index）中
- 同时搜索两个索引，然后合并结果
- 定期将辅助索引合并到主索引中
- 删除的处理：
  - 采用无效位向量(Invalidation bit-vector)来表示删除的文档
  - 利用该维向量过滤返回的结果，以去掉已删除文档

主辅索引合并中的问题

- 合并过于频繁
- 合并时如果正好在搜索，那么搜索的性能将很低
- 实际上:
  - 如果每个倒排记录表都采用一个单独的文件来存储的话，那么将辅助索引合并到主索引的代价并没有那么高
  - 此时合并等同于一个简单的添加操作
  - 但是这样做将需要大量的文件，效率显然不高
- 如果没有特别说明，本讲后面都假定索引是一个大文件
- 现实当中常常介于上述两者之间(例如：将大的倒排记录表分割成多个独立的文件，将多个小倒排记录表存放在一个文件当中……)

对数合并(Logarithmic merge)

- 对数合并算法能够缓解(随时间增长)索引合并的开销
  - → 用户并不感觉到响应时间上有明显延迟
- 维护一系列索引，其中每个索引是前一个索引的两倍大小
- 将最小的索引 (Z_0) 置于内存
- 其他更大的索引 ($I_0, I_1, \ldots$) 置于磁盘
- 如果 Z_0 变得太大 ($> n$), 则将它作为 I_0 写到磁盘中(如果 I_0 不存在)
- 或者和 I_0 合并(如果 I_0 已经存在)，并将合并结果作为I_1写到磁盘中(如果 I_1 不存在)，或者和I_1合并(如果 I_0 已经存在)，依此类推……

```
LMergeAddToken(indexes, Z_0, token)
 1  Z_0 ← Merge(Z_0, {token})
 2  if |Z_0| = n
 3    then for i ← 0 to ∞
 4         do if I_i ∈ indexes
 5              then Z_{i+1} ← Merge(I_i, Z_i)
 6                   (Z_{i+1} is a temporary index on disk.)
 7                   indexes ← indexes − {I_i}
 8              else I_i ← Z_i     (Z_i becomes the permanent index I_i.)
 9                   indexes ← indexes ∪ {I_i}
10                   Break
11         Z_0 ← ∅
```

```
LogarithmicMerge()
1  Z_0 ← ∅     (Z_0 is the in-memory index.)
2  indexes ← ∅
3  while true
4  do LMergeAddToken(indexes, Z_0, getNextToken())
```

对数合并的复杂度

- 索引数目的上界为 $O(\log T)$ (T 是所有倒排记录的个数)
- 因此，查询处理时需要合并$O(\log T)$个索引
- 索引构建时间为 $O(T \log T)$.
  - 这是因为每个倒排记录需要合并$O(\log T)$次
- 辅助索引方式： 索引构建时间为$O(T^2)$，因为每次合并都需要处理每个倒排记录
  - 假定每个辅助索引的大小为 a
  - $a + 2a + 3a + 4a + \ldots + na = a\frac{n(n+1)}{2} = O(n^2)$
- 因此，对数合并的复杂度比辅助索引方式要低一个数量级

本讲内容

- 两种索引构建算法: BSBI (简单) 和 SPIMI (更符合实际情况)
- 分布式索引构建: MapReduce
- 动态索引构建: 如何随着文档集变化更新索引

第5讲 索引压缩

Index compression

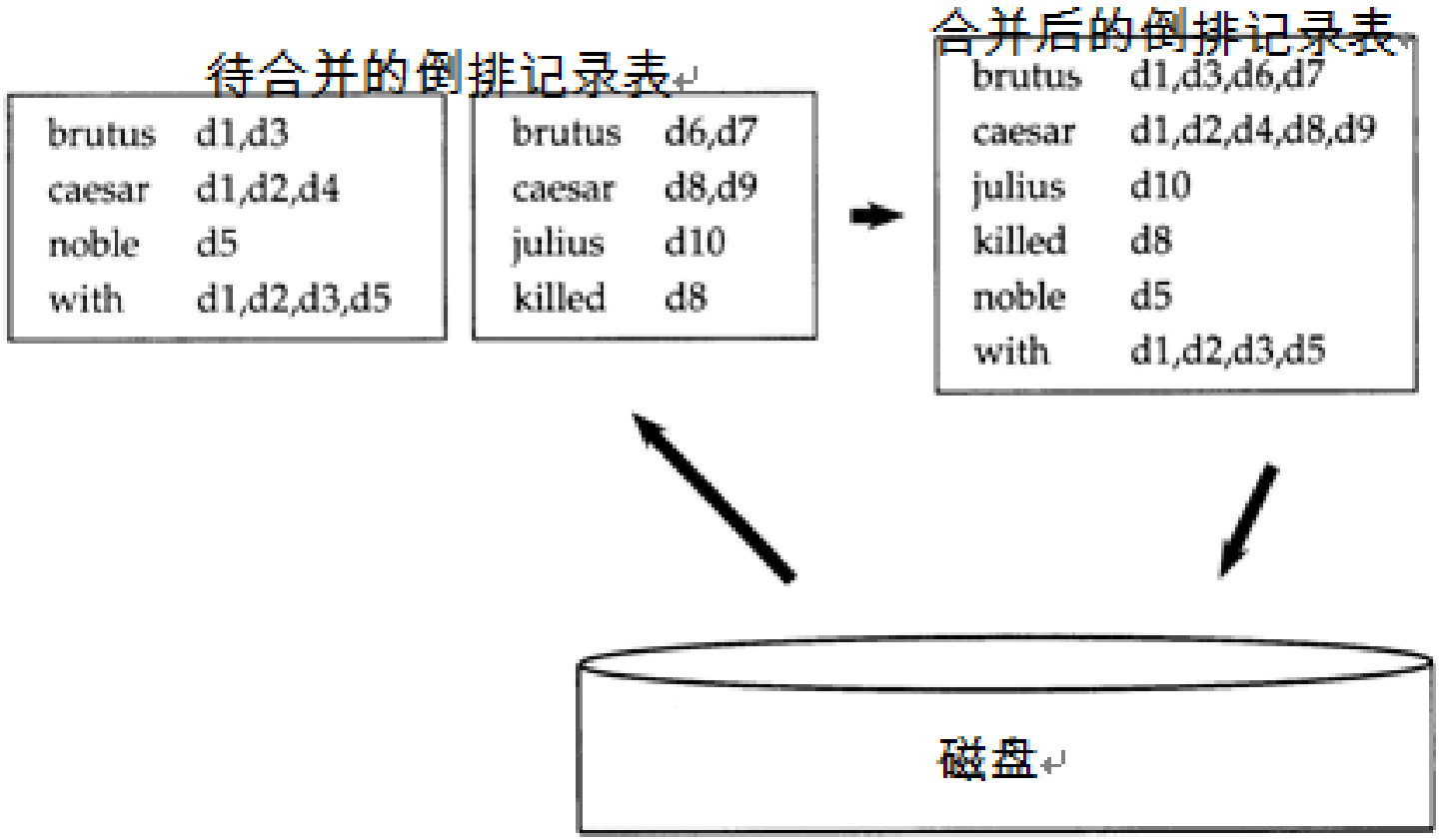
*节选自王斌老师《现代信息检索》课程的课件

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

基于块的排序索引构建算法BSBI



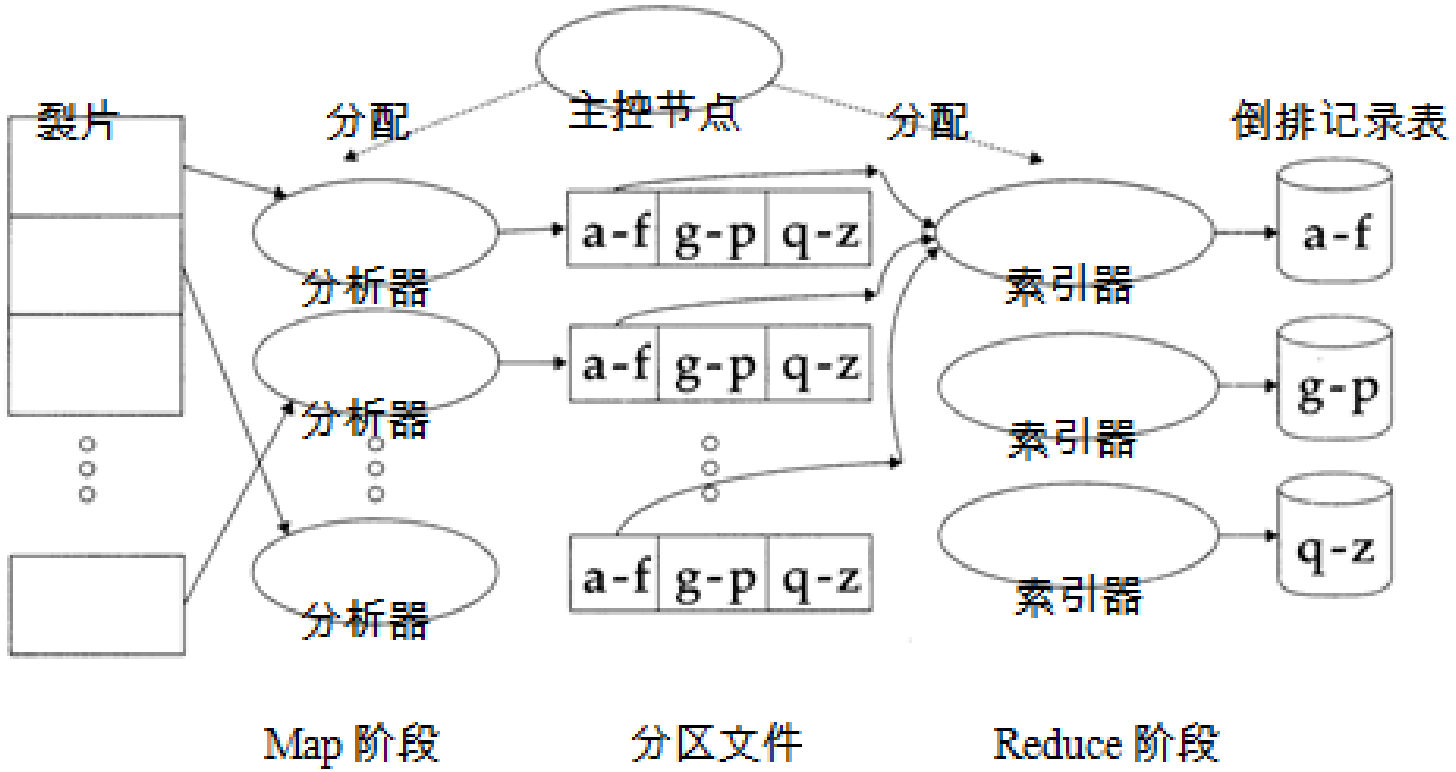
内存式单遍扫描索引构建算法SPIMI

- 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
```

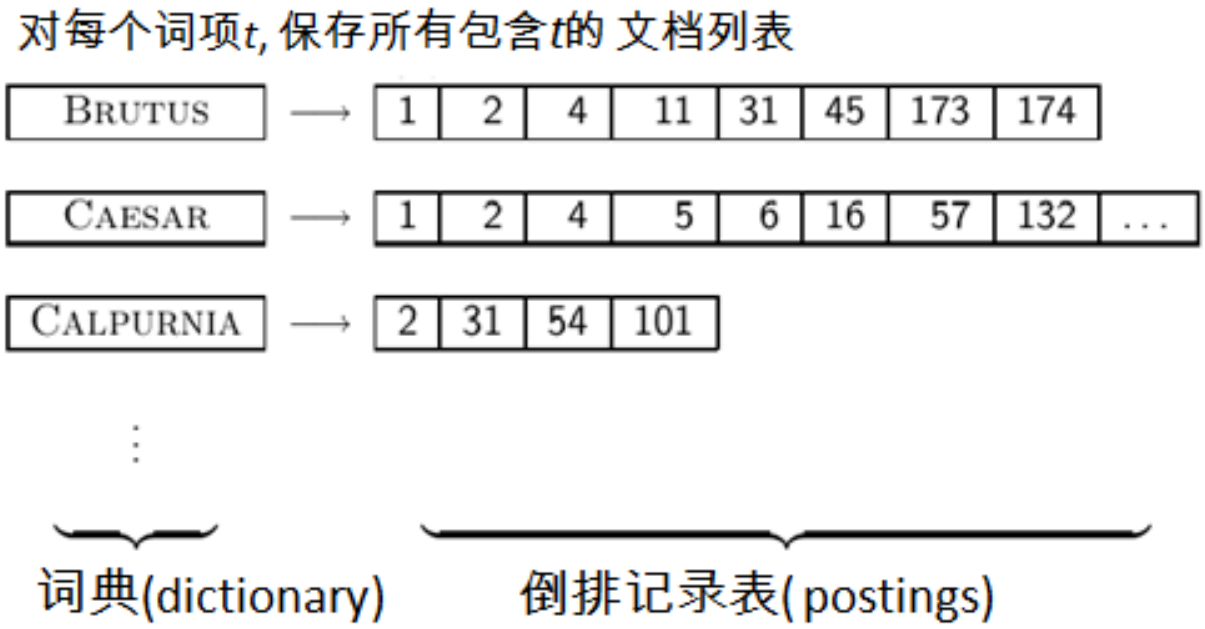
基于MapReduce的索引构建



动态索引构建：最简单的方法

- 在磁盘上维护一个大的主索引(Main index)
- 新文档放入内存中较小的辅助索引（Auxiliary index）中
- 同时搜索两个索引，然后合并结果
- 定期将辅助索引合并到主索引中

本讲内容



- 信息检索中进行压缩的动机
- 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？
- 词项统计量：词项在整个文档集中如何分布？

提纲

什么是压缩？

- 将长编码串用短编码串来代替
  - 1111111111111111➔ 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?

- 首先，需要考虑词典的存储空间
  - 词典压缩的主要动机: 使之能够尽量放入内存中
- 其次，对于倒排记录表而言
  - 动机: 减少磁盘存储空间，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
- 接下来，将介绍词典压缩和倒排记录表压缩的多种机制

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

N	文档数目	800,000
L	每篇文档的词条数目	200
M	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
T	无位置信息索引中的倒排记录数目	100,000,000

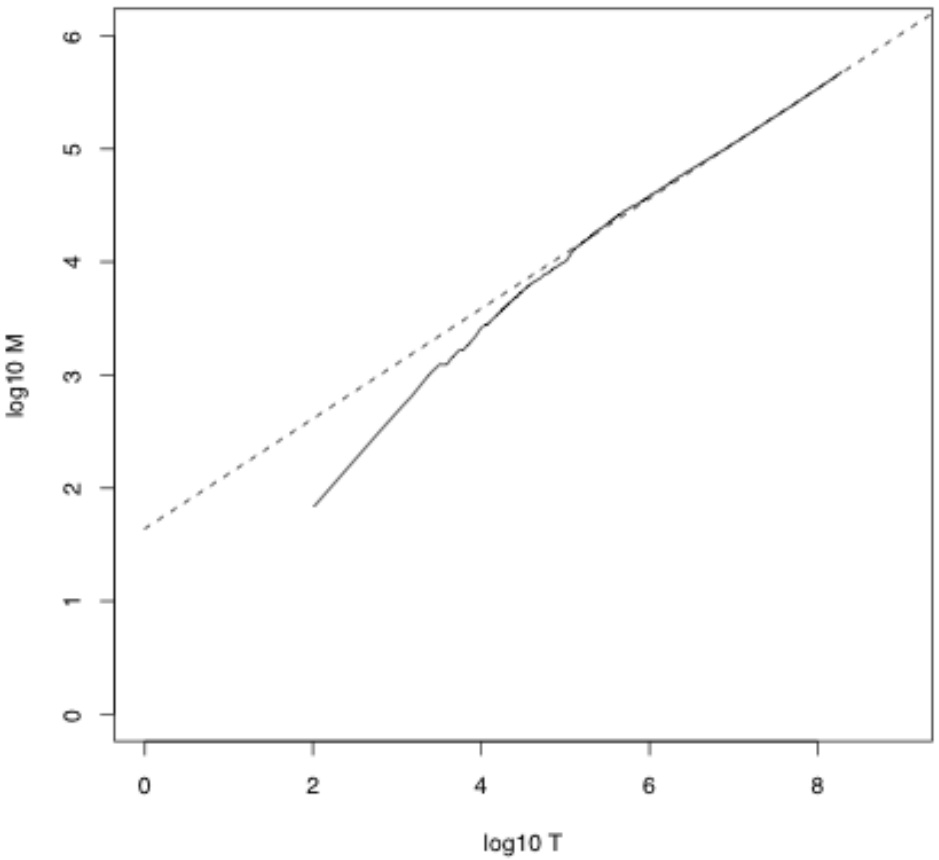
预处理的效果

	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

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



- 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
- Heaps定律提出了如下两点假设：

1. 随着文档数目的增加，词汇量会持续增长而不会稳定到一个最大值；

2. 大规模文档集的词汇量也会非常大。

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: 第i常见的词项的频率cf_i和$1/i$ 成正比

$$\text{cf}_i \propto \frac{1}{i}$$

- cf_i 是文档频率(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)

Zipf定律

- Zipf定律: 第*i*常见的词项的频率cf_i和1/*i* 成正比

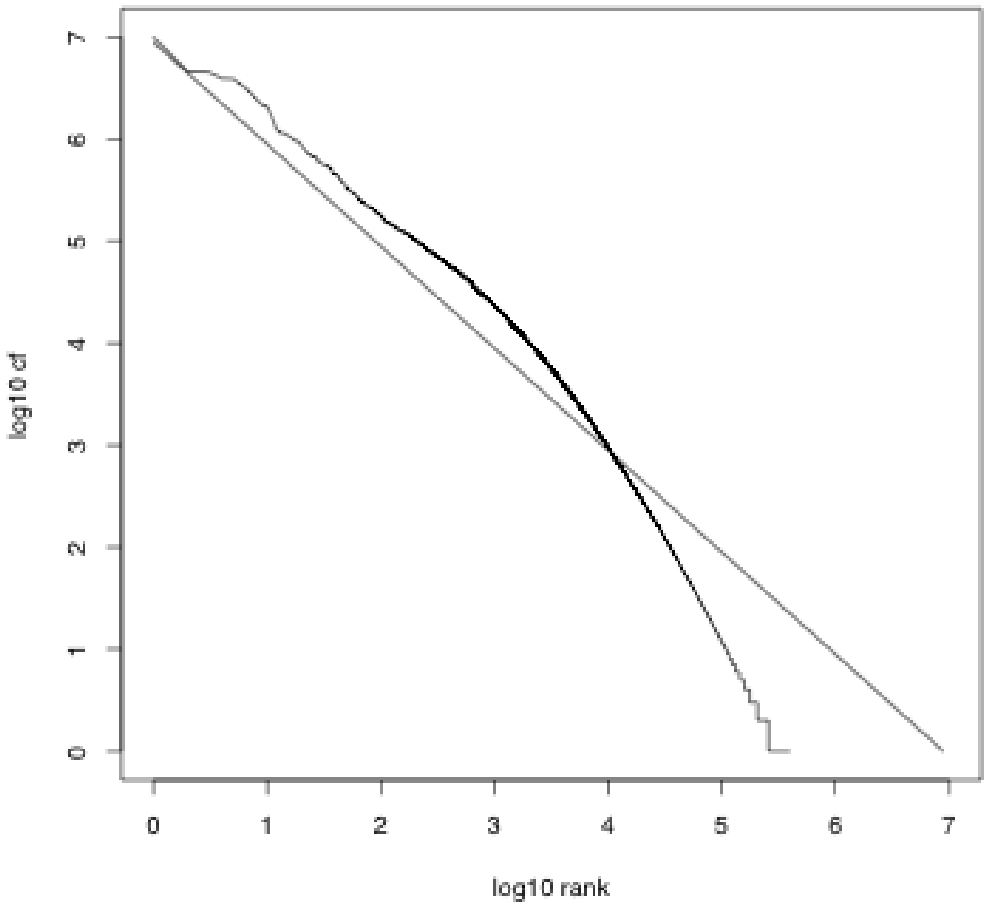
$$\mathrm{cf}_i \propto \frac{1}{i}$$

- cf_i 是文档频率(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 $\mathrm{cf}_2 = \frac{1}{2}\mathrm{cf}_1 \quad \ldots$

. . . 第三常见的词项 (*and*) 出现次数为 $\mathrm{cf}_3 = \frac{1}{3}\mathrm{cf}_1$

- 另一种表示方式: $\mathrm{cf}_i = ci^k$ 或 $\log \mathrm{cf}_i = \log c + k \log i \quad (k = -1)$
- K=-1情况下幂定律(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

词典压缩

- 相对于倒排记录表，词项较小
- 但是为了检索效率，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

词项	文档频率	指向倒排记录表的指针
a	656 265	→
aachen	65	→
…	…	…
zulu	221	→

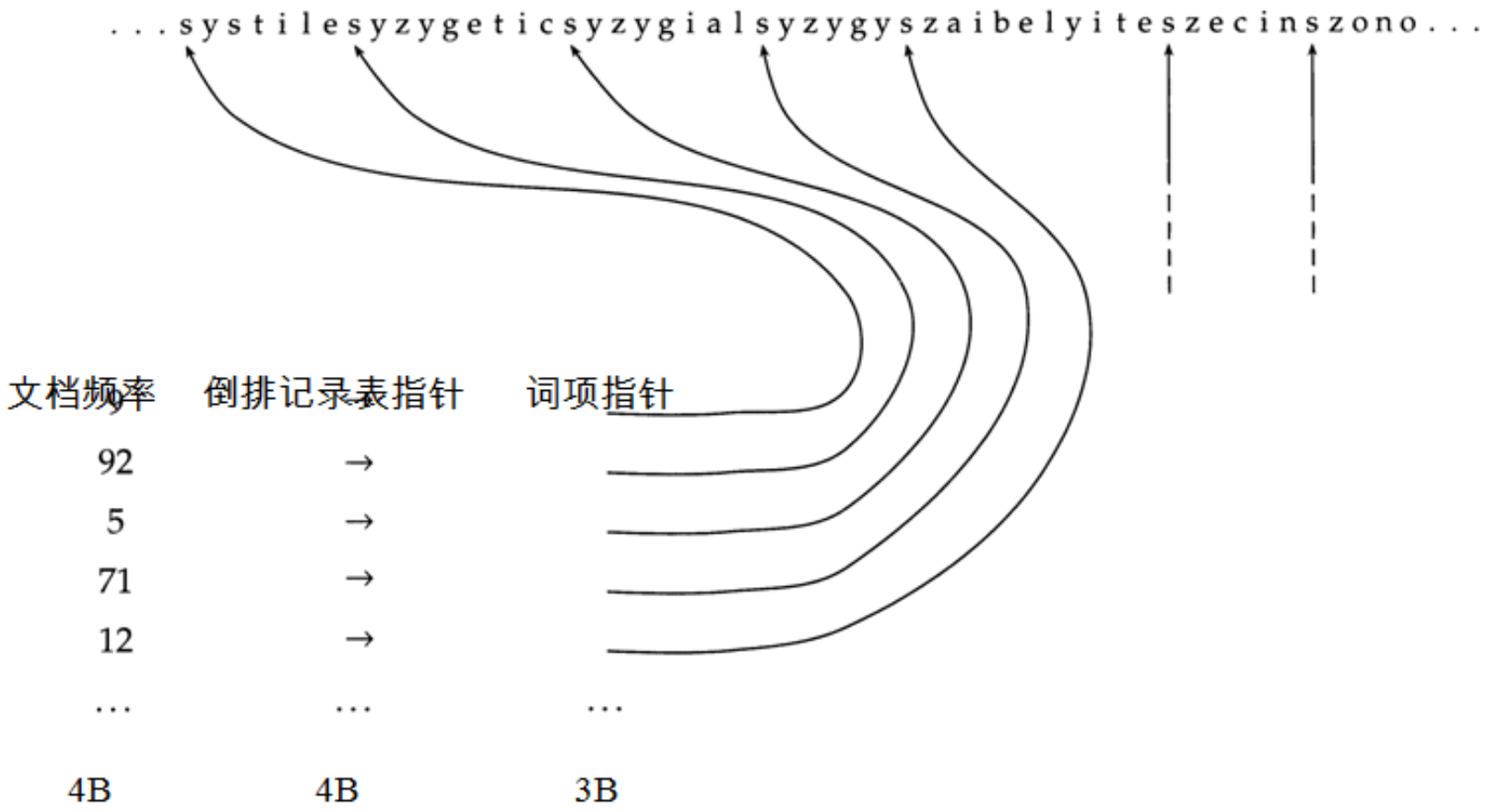
空间需求： 20 字节 4 字节 4 字节

对Reuters RCV1语料: (20+4+4)*400,000 = 11.2 MB

定长方式的不足

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

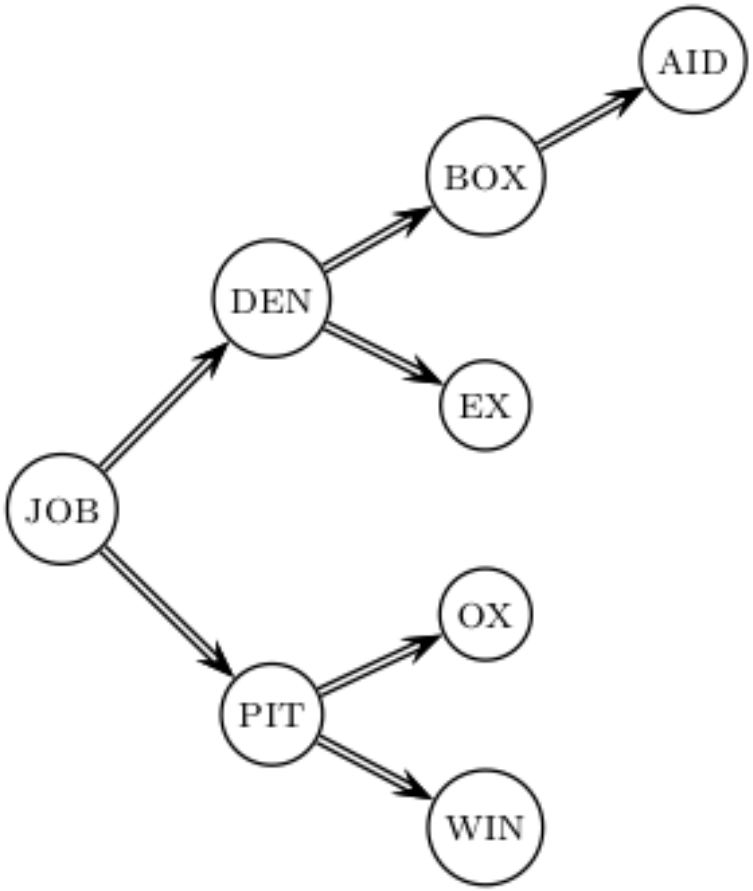
- 每个词项的词项频率需要4个字节
- 每个词项指向倒排记录表的指针需要4个字节
- 每个词项平均需要8个字节
- 指向字符串的指针需要3个字节 (8*400000个位置需要 $\log_2$（8 * 400000） < 24 位来表示)
- 空间消耗: 400,000 × (4 +4 +3 + 8) = 7.6MB (而定长数组方式需要11.2MB)

单一字符串方式下按块存储

按块存储下的空间消耗

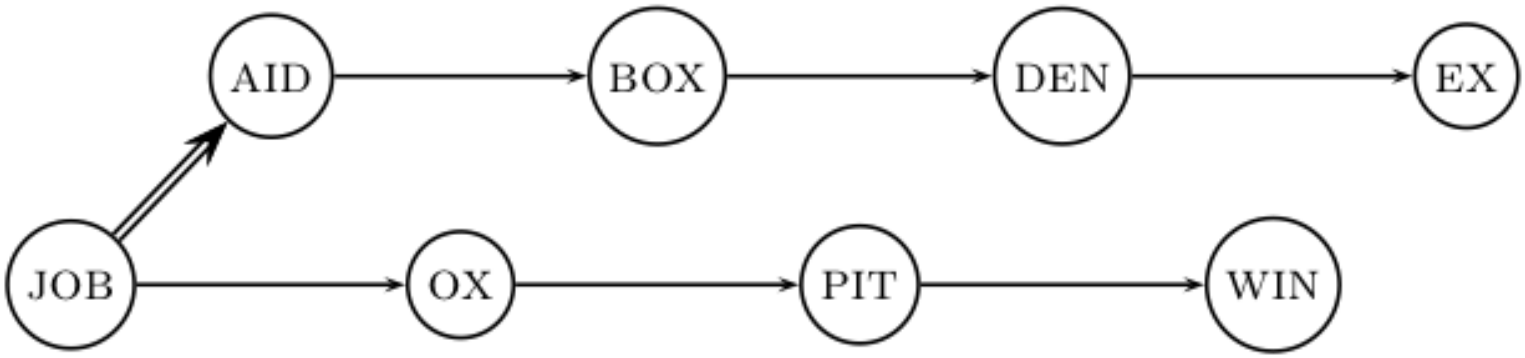
- 如果不按块存储，则每4个词项指针将占据空间4 × 3=12B
- 现在按块存储，假设块大小k=4，此时每4个词项只需要保留1个词项指针，但是同时需要增加4个字节来表示每个词项的长度，此时每4个词项需要3+4=7B
- 因此，每4个词项将节省12-7=5B
- 于是，整个词典空间将节省40,000/4*5B=0.5MB
- 最终的词典空间将从7.6MB压缩至7.1MB

不采用块存储方式下的词项查找



假定图中每个后续词项的出现概率都相等的话，那么在未压缩的词典中查找的平均时间为(0 + 1 + 2 + 3 + 2 + 1 + 2 + 2) / 8≈ 1.6 步。

采用块存储方式下的词项查找: 稍慢



在图中所示的结构中的平均查找时间为
(0 + 1+ 2 + 3 + 4 + 1 + 2 + 3) / 8 = 2 步

前端编码(Front coding)

每个块当中 (k = 4)，会有公共前缀 . . .可以采用前端编码方式继续压缩

8automata8automate9automatic10automation

⇓

8automat∗a1⋄e2⋄ic3⋄ion

- 前端编码示意图。图中多个连续词项具有公共前缀automat，那么在前缀的末尾用“ ∗” 号标识，在后续的词项中用“ ◇” 表示该前缀。和前面一样，每个词项的前面第一个字节存储了该词项的长度

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

提纲

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

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	2	…
computer	文档ID	…	283 047	283 154	283 159	283 202	…
	文档ID间距			107	5	43	…
arachnocentric	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
  - 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：将低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）

VB 编码算法

```
VBENCODENUMBER(n)
1  bytes ← ⟨⟩
2  while true
3  do PREPEND(bytes, n mod 128)
4     if n < 128
5        then BREAK
6     n ← n div 128
7  bytes[LENGTH(bytes)] += 128
8  return bytes
```

```
VBENCODE(numbers)
1  bytestream ← ⟨⟩
2  for each n ∈ numbers
3  do bytes ← VBENCODENUMBER(n)
4     bytestream ← EXTEND(bytestream, bytes)
5  return bytestream
```

VB编码的解码算法

```
VBDecode(bytestream)
 1  numbers ← ⟨⟩
 2  n ← 0
 3  for i ← 1 to Length(bytestream)
 4  do if bytestream[i] < 128
 5        then n ← 128 × n + bytestream[i]
 6        else n ← 128 × n + (bytestream[i] − 128)
 7             Append(numbers, n)
 8             n ← 0
 9  return numbers
```

γ编码

- 另外一种变长编码是基于位的编码
- γ 编码是其中最出名的一种
- 首先，在介绍γ编码之前先介绍一元码(unary code)
- 一元码：
  - 将 n 表示成 n 个1和最后一个0
  - 比如： 3的一元码是 1110
  - 40的一元码是 1111111111111111111111111111111111111110
  - 70的一元码是：

1111111111111111111111111111111111111111111111111111111111111110

γ编码

- 将G 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 → 1101 → 101 = 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	1111111110	1110	001	1110,001
13		1110	101	1110,101
24		11110	1000	11110,1000
511		111111110	11111111	111111110,11111111
1025		11111111110	0000000001	11111111110 ,0000000001

ϒ编码的长度

- 偏移部分是 [log2 G] 比特位
- 长度部分需要 [log2 G] + 1 比特位
- 因此，全部编码需要2[log2 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log2 G—实际中并非如此)
  - (assuming the frequency of a gap G is proportional to log2 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

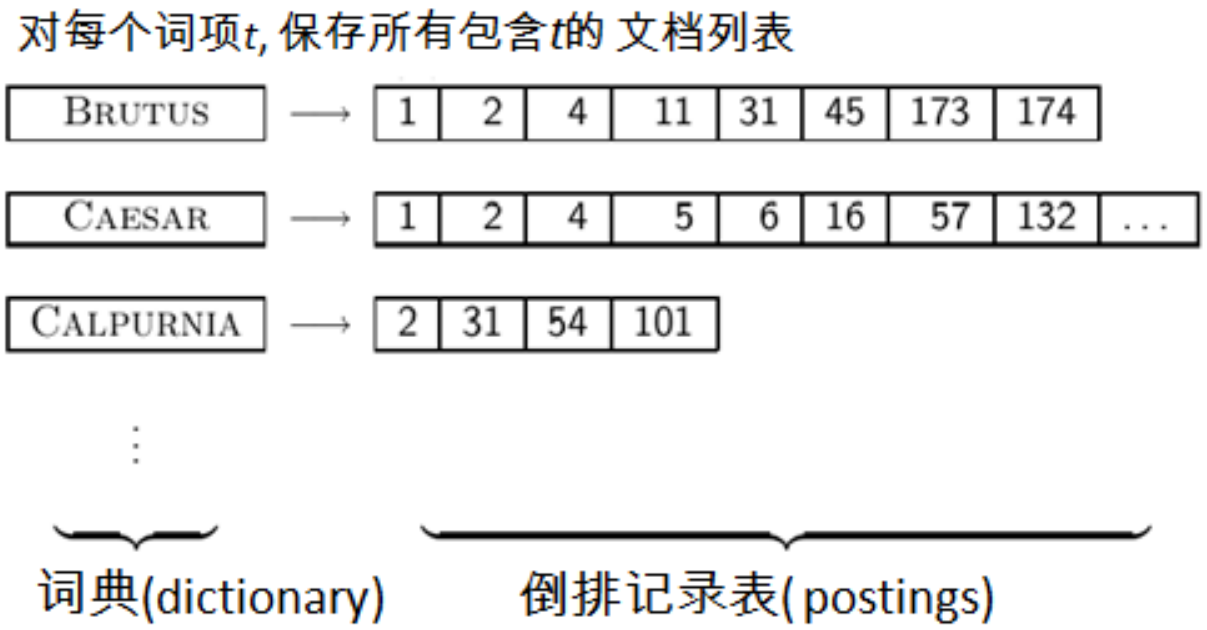
Reuters RCV1索引压缩总表

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

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

本讲小结



- 信息检索中进行压缩的动机
- 倒排索引中词典部分如何压缩？长字符串方法及改进
- 倒排索引中倒排记录表部分如何压缩？可变字节码、 γ编码
- 词项统计量: 词项在整个文档集中如何分布？ 两个定律