索引构建与压缩
Index Construction
Index Compression

索引构建

索引构建

- 索引构建，Index Construction 或 Indexing
- 构建索引的程序或计算机称倒排器（索引器）Indexer

- 思考如下问题：
  - 怎样建立一个索引？
  - 对于给定的计算机内存，可以采用怎样的索引构建策略?

索引构建

- 硬件基础
- 语料库介绍
- 索引构建算法
- 分布式索引构建
- 动态索引

硬件基础：存储能力

- IR系统的服务器通常“数GB”甚至“数百GB”的内存。
- 其可用磁盘空间大小一般比内存大小高几个(2-3)数量级(TB级别)。
- 容错控制代价非常昂贵：使用许多台常规服务器要比使用一台容错服务器便宜得多。

硬件基础：计算机I/O能力(2007)

- 访问内存数据比访问磁盘数据快得多。
- 磁盘寻道：磁头移到数据所在的磁道需要一段时间，寻道期间并不进行数据的传输。
- 因此：从磁盘到内存传输一个大数据块要比传输很多小的数据块快的多。
- 磁盘读写操作是基于**块**的：从磁盘读取一个字节和读取一个数据块所耗费的时间可能一样多。
- 块大小：8KB – 256KB

典型硬件性能参数(2007年水平)

符号	含义	值
s	平均寻道时间	$5ms = 5 \times 10^{-3}s$
b	每个字节的传输时间	$0.02\ \mu s = 2 \times 10^{-8}s$
	处理器时钟频率	$10^9\ s^{-1}$（也就是GHz）
p	底层操作时间 （如单词的比较或者交换）	$0.01\mu s = 10^{-8}s$
	内存大小	几个GB
	磁盘空间大小	1TB或者更多

索引构建

- 硬件基础
- 语料库介绍
- 索引构建算法
- 分布式索引构建
- 动态索引

RCV1语料库:样例文档集

- 为了阐述本课程的许多要点，《莎士比亚全集》作为样例文档集远远不够。
- Reuters-RCV1文档集
  - 不是真正的足够大，但是公开的，一个更为合理的样例。
  - 将使用路透社的RCV1文档集作为“可扩展的索引构建算法”的样例。
  - 该文档集由一年的路透社新闻组成(1995-1996)。

Reuters-RCV1语料：统计数据

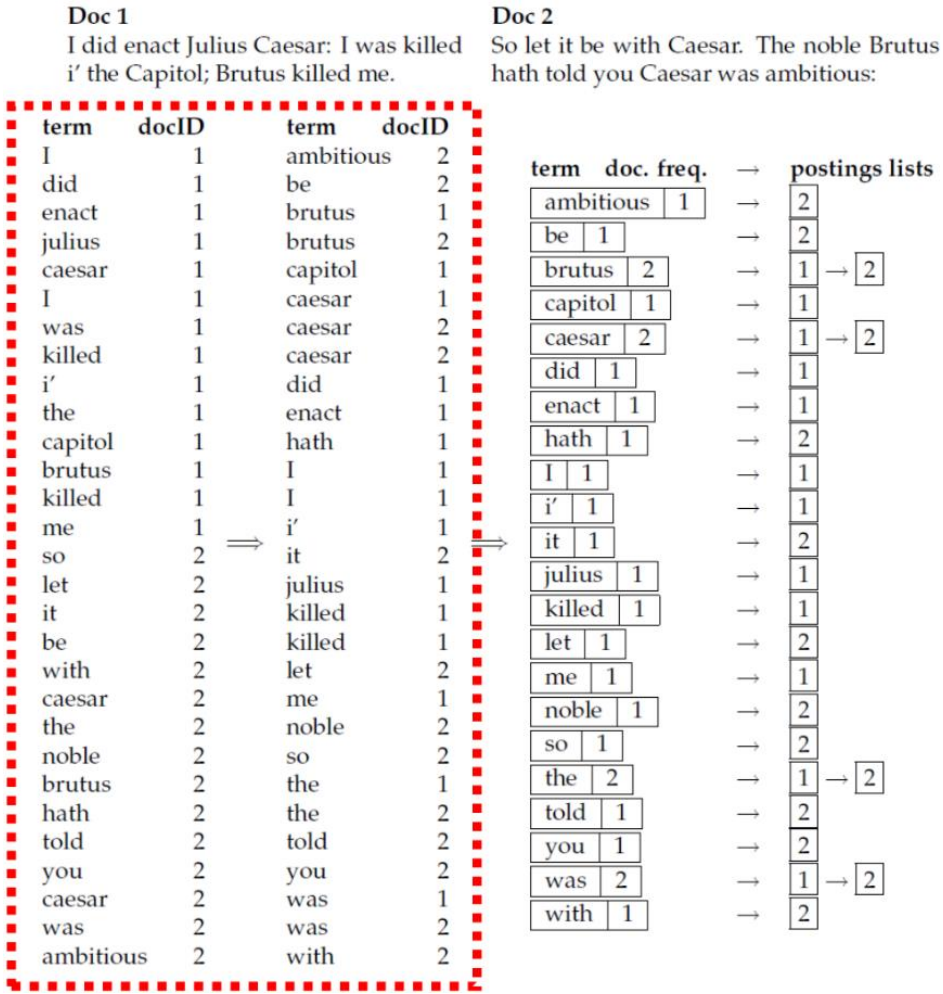
符号	含义	值
N	文档总数	800 000
L_{ave}	每篇文档的平均词条(Token)数目	200
M	词项(Term)总数	400 000
	每个词条的平均字节数(含空格和标点符号)	6
	每个词条的平均字节数(不含空格和标点符号)	4.5
	每个词项的平均字节数	7.5
T	词条(Token)总数	160 000 000

词条就是tokenize后的
而词项是真正索引的，包括单复数变化，去停用词等等

200*800000

Reuters-RCV1语料：索引构建中的临时文件

- 文档ID需32bit=4Byte
- 词条ID需32bit (总共约1亿词条)
- 存储所有的“词条ID-文档ID”需要
  - 约100,000,000 * (32+32) = 6,400,000,000bits = 800,000,000Bytes = 0.8GB 存储空间
- 需要对0.8GB的ID对进行排序！！！
- 而实际语料库要比RCV1更大



扩展索引构建

- 在内存中进行索引构建并不能扩展。
- 怎样才能对大型的语料库构建索引？
- 考虑到我们刚刚了解的硬件约束条件。。。
  - 内存，硬盘，速度等等。

基于排序的索引构建算法

- 在建立索引过程中，需要依次分析所有的文档。
  - 索引构建过程中，不能很容易地利用压缩技巧(即使可以，也会非常复杂)。
- 只有分析完所有的文档，最终的倒排记录表才会完整
- 每一个<词项，文档，频数>对占用**12**字节，对于大型语料库则需要非常大的空间。
- 在**RCV1**文档集中，倒排记录总数**T=100,000,000**
  - 仍然可以在内存中对所有词项ID-文档ID对进行排序。
  - 但是通常的语料库会大很多，例如：《纽约时报》提供了一份包含超过150年新闻的索引文件。
- **因此：需要在硬盘中存储中间的结果。**

索引构建

- 硬件基础
- 语料库介绍
- 索引构建算法
- 分布式索引构建
- 动态索引

索引构建算法

- 基于块的排序索引构建算法
  - 面向静态文档集
  - 单机
- 内存式单遍扫描索引构建算法
- 分布式索引构建算法
- 动态索引构建算法

在硬盘中采用同样的算法？

- 对于大型的语料库，能在硬盘而不是内存中采用同样的索引构建算法吗？
- 答案是“**NO**”：在硬盘中排序T=100,000,000条记录太慢了——需要很多次的磁盘寻道。
- 需要一个外部排序算法。

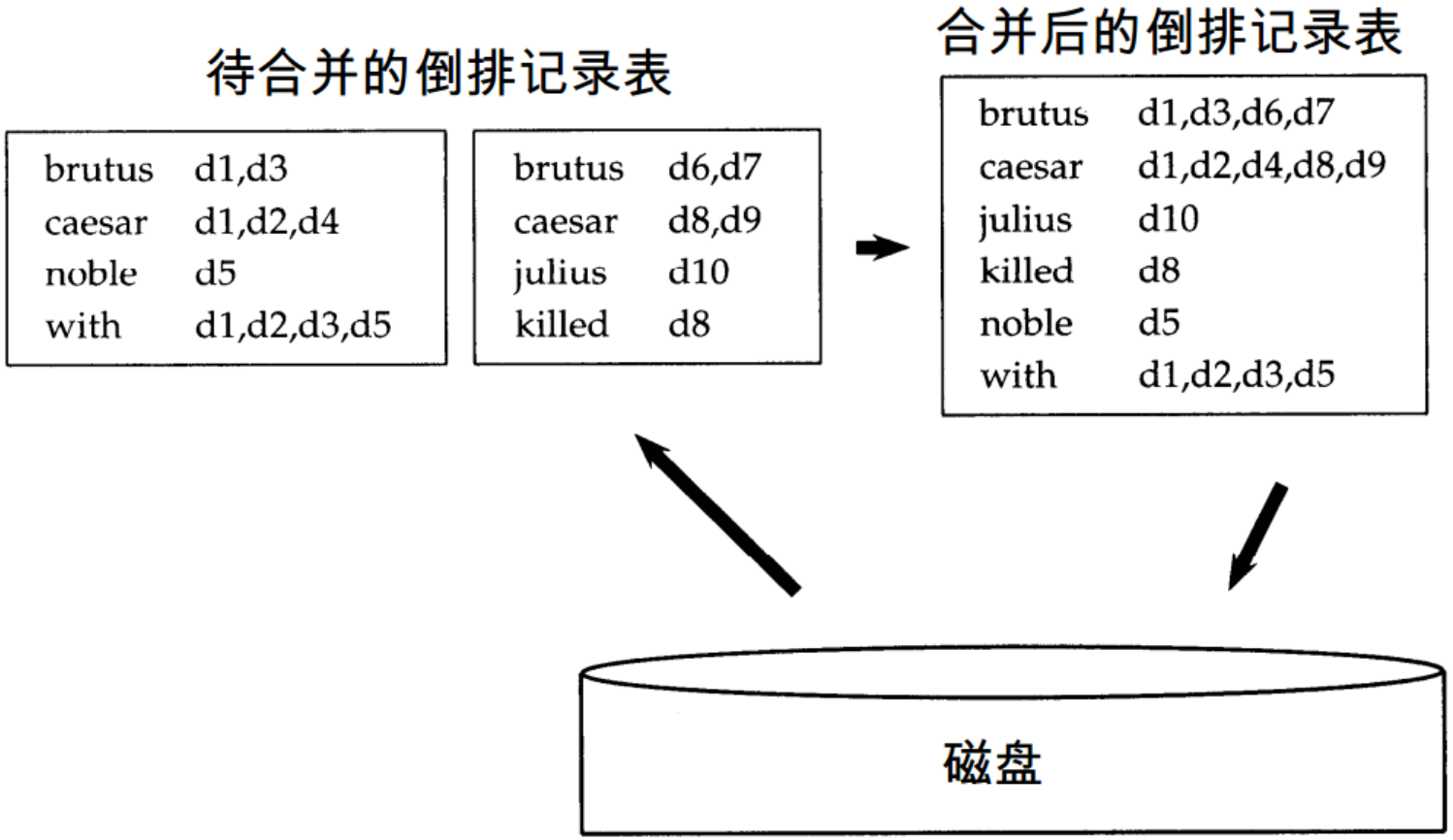
瓶颈

- 依次对文档进行分析并建立倒排记录项<Term, DocID>。
- 根据**词项**对所有倒排记录项进行排序
  - 然后在词项内再根据**文档ID**进行二次排序。
- 由于需要随机的磁盘寻道，在硬盘中进行排序非常慢——必须排序T=1亿条记录。

假如每次比较需要2次的磁盘寻道，对N条记录进行排序需要$N \log_2 N$ 次比较，我们需要花费多少的时间？

基于块的排序索引算法
BSBI: Blocked sort-based Indexing

- 基本思想
  - 对每一个块都生成倒排记录，并排序，写入硬盘
  - 然后将这些块合并成一个长的排好序的倒排记录。



BSBI(基于块的排序索引算法)
(需要较少的磁盘寻道次数)

- 每条数据占用12字节(4+4+4) (词项，文档，频数)
- 这些数据是在分析文档时生成
- 需要对100M条这样12字节的数据进行排序
- 定义一个块~10M大小的数据
  - 可以很容易地加载数个这样的块数据到内存中
  - 开始加载10个这样的块数据

- **100M数据的排序→排序10块10M的数据**
- 必须在硬盘上直接排序→在内存中排序(10M)
- 带来的问题：**需要合并10个排序后的结果**

BSBI(基于块的排序索引算法)

- $f_1, f_2, \ldots$合并为f_{merged}

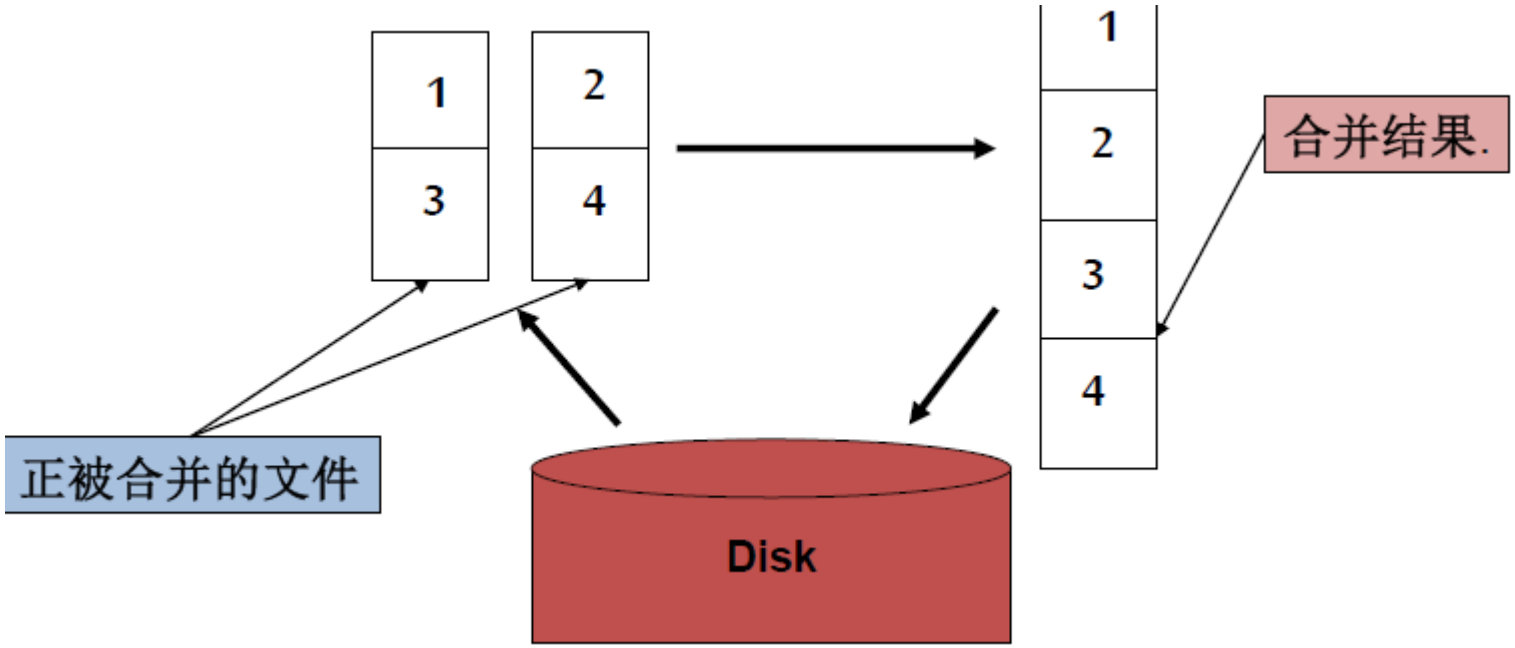
BSBIndexConstruction()

```
1  n ← 0
2  while (all documents have not been processed)
3  do n ← n + 1
4      block ← ParseNextBlock()        写满块
5      BSBI-Invert(block)              构建块索引
6      WriteBlockToDisk(block, f_n)    写块索引文件
7  MergeBlocks(f_1, ..., f_n; f_merged)   合并索引文件
```

- 在内存中处理,累积放满固定的块,排序后写入硬盘f_i
- 合并所有块索引文件f_i成一个

如何合并排序结果？

- 可以进行二分合并，产生一个$\log_2 10$，4层的合并树。
- 在每一层中，读入对应的块文件到内存中，合并倒排记录表，合并结果写回磁盘中。



如何合并排序结果？

- 一个n-路的合并会更加高效，可以同时读取所有的数据块。
- 内存中维护
  - 为10个块准备的读缓冲区
  - 一个为最终合并索引准备的写缓冲区
  - 这样就不会因为硬盘寻道而浪费大量的时间了。

基于BSBI排序的算法存在的问题

- 假设：能够将词典存入内存中。
- 需要该词典(动态增长)去查找任一词项和词项ID之间的对应关系。
- 事实上，可以采用<词项，文档ID>对来代替<词项ID，文档ID>对。
  - 每个词项的平均字节数=7.5
- …但是中间文件会变的非常的大。

（→一个可拓展的，但效率非常低的索引构建算法）

SPIMI:内存式单遍扫描索引算法

- SPIMI:Single-pass in-memory indexing
- **核心思想1：**为每个块单独生成一个词典—不需要维护全局的<词项，词项ID>映射表。
- **核心思想2：**不进行排序。有新的<词项，文档ID>对时直接在倒排记录表中增加一项。
- 根据这两点思想，可以为每个块生成一个完整的倒排索引。
- 然后将这些单独的索引合并为一个大的索引。

SPIMI：压缩

- 压缩技术将会使SPIMI算法更加高效。
  - 压缩词项
  - 压缩倒排记录表
- 见下一部分：索引压缩

索引构建

- 硬件基础
- 语料库介绍
- 索引构建算法
- 分布式索引构建
- 动态索引

分布式索引构建 Distributed indexing

- Web大规模的索引构建
  - 必须使用一个分布式的计算机集群
- 这些计算机都是故障频发的
  - 可能会在任意时刻失效
- 如何开发这样一个计算机集群？

Google数据中心

- Google数据中心主要是由商用计算机组成
- 数据中心分布在世界各处
- 估计：总共一百万台服务器
  - （Estimated by prof. Koomey from Stanford 2011）
- 估计：Google每季度就会安装100,000台服务器
  - 根据它每年2-2.5亿美元的开销

Google数据中心

- 假如在一个包含1000个节点的非容错系统中，每个节点的正常运行概率为99.9%，那么这个系统的正常运行概率为多少？
  - 答案是：36.8% ($99.9\%^{1000}$)
- 可以试着计算一下：对于一个一百万台计算机的集群，每分钟会有多少台服务器宕机。

分布式索引构建

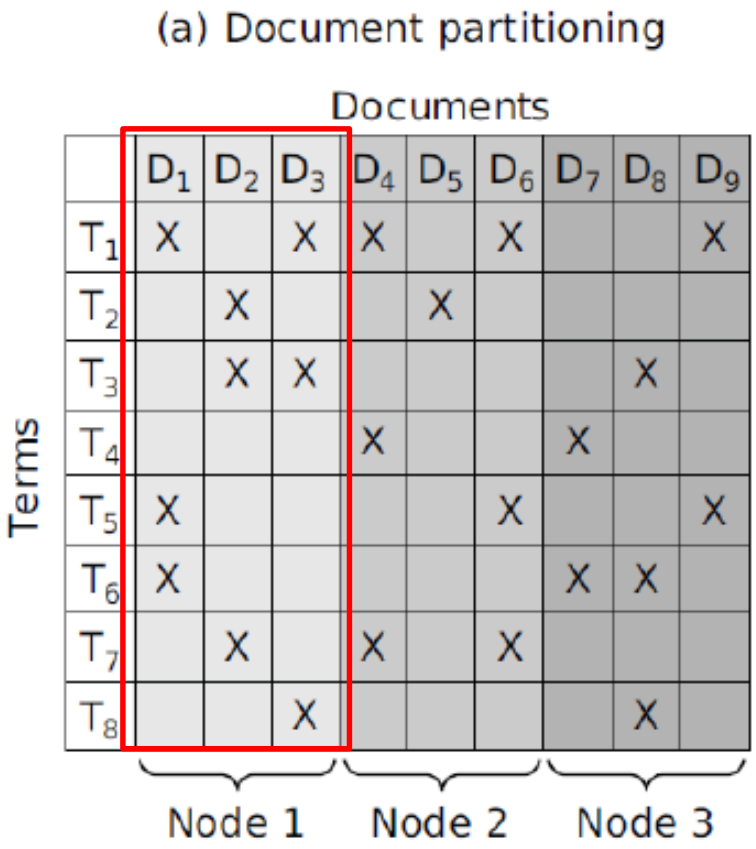
- 利用集群中的主控节点来指挥索引构建工作。
  - 假设主控节点是“安全”的。
- 将索引构建过程分解成一组并行的任务。
- 主控计算机从集群中选取一台空闲的机器并将任务分配给它。

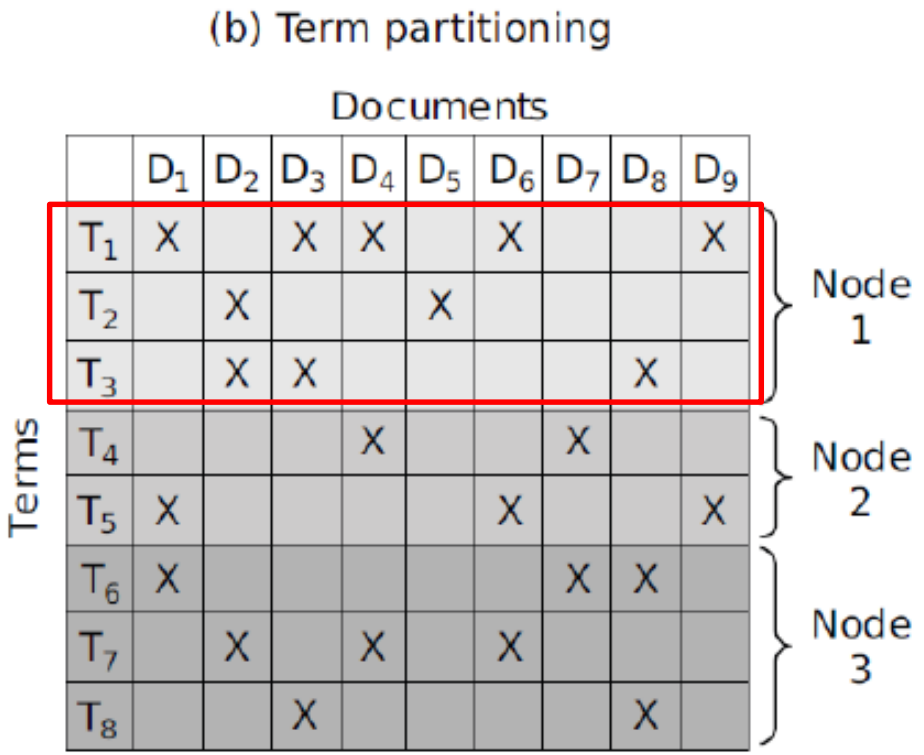
并行任务

- 采用两组不同的并行任务
  - Parsers 分析器
  - Inverters 倒排器
- 首先，将输入文档集分割成*n*个数据片
  - 每个数据片就是一个文档子集(与BSBI/SPIMI算法中的数据块相对应)

文档集分割

- 两种分割方法
  - 基于文档的分割
  - 基于词项的分割





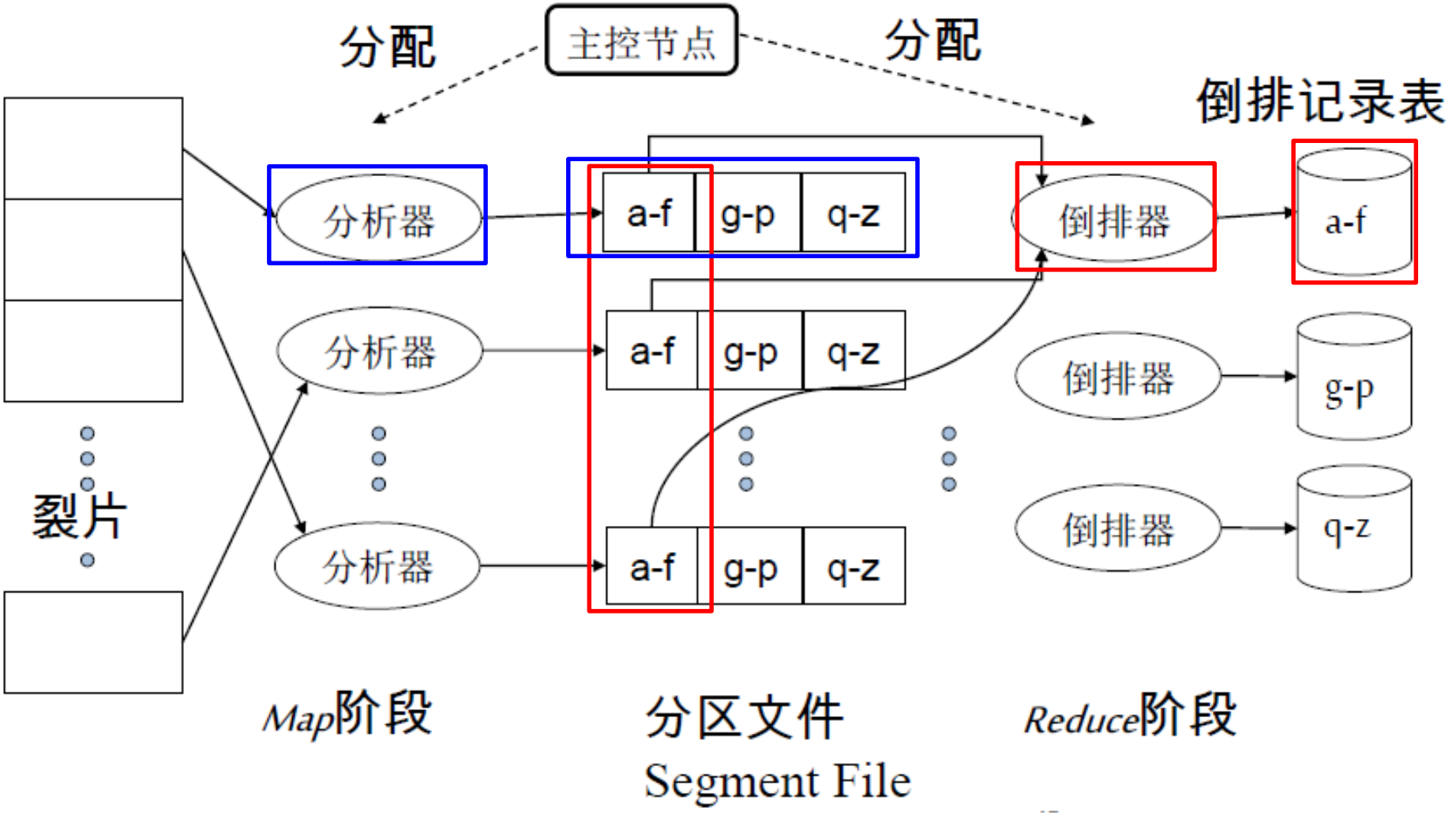
分析器 Parsers

- 主节点将一个数据片分配给一台空闲的分析服务器
- 分析器依次读取文档并生成 <词项,文档>对
- 分析器将这些<词项,文档>按照词项分成***j***个段
- 每一段是按照词项首字母划分的一个区间
  - (例如：***a-f, g-p, q-z***)-这里 j=3
- 然后可以进行索引的倒排

倒排器Inverters

- 对于一个词项分区，倒排器收集所有的<词项, 文档>对 (也就是“倒排记录”)
- 排序，并写入最终的倒排记录表

数据流


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
倒排器
倒排器
倒排器
a-f
g-p
q-z
裂片
Map阶段
分区文件
Segment File
Reduce阶段

MapReduce

- 刚刚所讲的索引构建算法是MapReduce的一个应用
- MapReduce(Dean and Ghemawat 2004)是一个稳定的并且概念简单的分布式计算架构
  - 不需要自己再对分布式部分书写代码
- Google索引系统(ca.2002)由各个不同的阶段组成，每个阶段都是MapReduce的一个应用

- 索引构建只是其中的一个阶段
- 另一个阶段是：将基于词项划分的索引表转换成基于文档划分的索引表
  - 基于词项划分的：一台机器处理所有词项的一个子区间
  - 基于文档划分的：一台机器处理所有文档的一个子区间
- 在本课程的**Web**搜索部分会讲到，大部分搜索引擎都是采用基于文档划分的索引表
  - 优点：更好的负载平衡等等

采用MapReduce的索引构建架构

- **Map和Reduce函数的架构**
  - **Map**:输入→list(k, v) Reduce:(k, list(v))→输出
- 索引构建中上述架构的实例化
  - Map: Web文档集 → list(词项ID,文档ID)
  - Reduce: (<词项ID1, list(文档ID)>, <词项ID2,list(文档ID)>,…) → (倒排记录表1,倒排记录表2,…)
- 教材中索引构建的例子
  - Map: d_1: C came, C c'ed. d_2: C died. → <C,d_1>,<came,d_1>,<C,d_1>, <c'ed,d_1>, <C,d_2>,<died,d_2>)
  - Reduce: (<C,(d_1,d_1, d_2)>,<died,(d_2)>,<came,(d_1)>,<c'ed,(d_1)>)

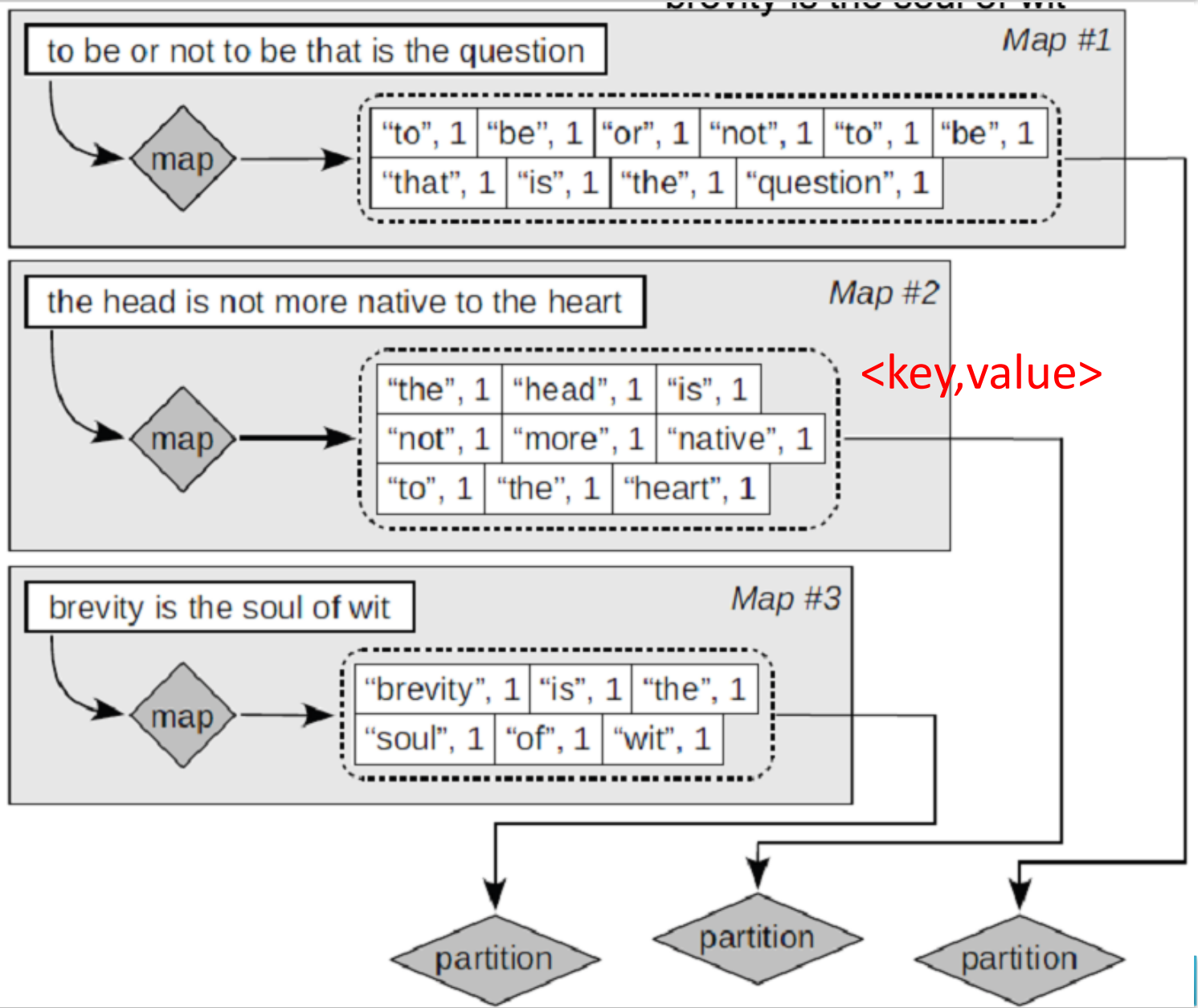
→(<C,(d_1:2,d_2:1)>,<died,(d_2:1)>,<came,(d_1:1)>,<c'ed,(d_1:1)>)

一个简单的例子：Map阶段

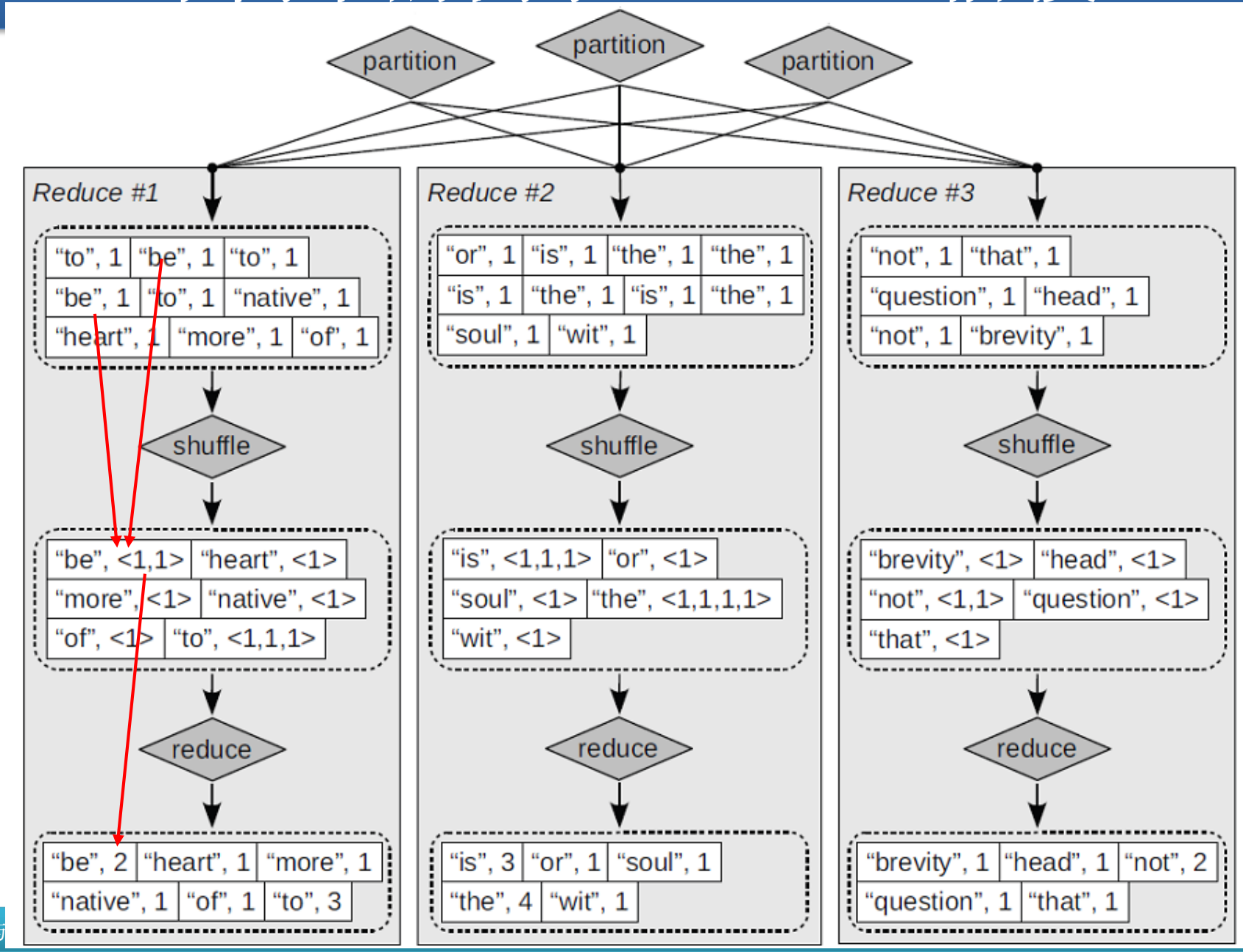
莎士比亚《哈姆雷特》

一个文档

- To be, or not to be: that is the question
- the head is not more native to the heart
- brevity is the soul of wit



一个简单的例子：Reduce阶段



索引构建

- 硬件基础
- 语料库介绍
- 索引构建算法
- 分布式索引构建
- 动态索引

动态索引构建方法

- 迄今为止，我们都假设文档集是静态的
- 但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
  - 新的词项加入到词典中

（1）最简单的索引更新方法

- **周期性索引重构**
  - 建立新索引的同时，旧索引继续工作
- 条件
  - 更新次数不是很多
  - 能够接受对新文档检索的一定延迟（重构之前新文档检索不到）
  - 有足够的资源进行重构

（2）方法2

- 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(位于内存)
- 检索可以同时遍历两个索引并将结果合并
- 删除
  - 文档的删除记录在一个无效位向量(invalidation bit vector)中
  - 在返回结果前利用它过滤掉已删除文档
- 定期地，将辅助索引合并到主索引中
- 文档更新通过先删除后插入方式实现

主索引与辅助索引存在的问题

- 频繁的合并 — 带来很大的开销
- 合并过程效率很低
  - 如果每个词项的倒排记录表都单独成一个文件，那么合并主索引和辅助索引将会很高效
  - 合并将是一个简单的添加操作
  - 但需要非常多的倒排文件 — 对文件系统来说是低效的
- 以后课程中都假设：索引是一个大的文件
- 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；并对那些长度为1的索引记录表进行合并)

对数合并

- 维护一系列的索引$I_0, I_1, I_2, \ldots$，每个都是前一个的两倍大小2^0*n, 2^1*n, 2^2*n, …。n是辅助索引Z_0的大小
- 辅助索引Z_0存储在内存中
- 将较大的那些($I_0, I_1, \ldots$)存储在磁盘中
- 当Z_0达到上限n时，将它写入磁盘的I_0中(此时$I_0=2^0*n$)
- 当Z_0下一次达到上限时，它会和I_0合并，生成Z_1 (大小2^1*n)
  - 此时，如果I_1不存在，存储到I_1中
  - 如果I_1已存在，则Z_1与I_1合并成Z_2 (大小2^2*n)
  - 此时，如果I_2不存在，存储到I_2中
  - 如果I_2已存在，则Z_2与I_2合并成Z_3 (大小2^2*n)
  - 以此类推…

拥有多个索引产生的问题

- 全局统计信息很难得到
- 例如：对于拼写校正算法，得到几个校正的备选词后，选择哪个呈现给用户？
  - 可以返回具有最高选中次数的那些
- 对于多个索引和无效位向量，怎样维护那些拥有最高次数的结果？
  - 一个可能的方法：除了主索引的排序结果，忽略其它所有的索引
- 事实上，采用对数合并方法，信息检索系统的各个方面，包括索引维护，查询处理，分布等等，都要复杂的多

搜索引擎中的动态索引

- 现在所有的大型搜索引擎都采用动态索引
- 它们的索引经常增加和改变
  - 新的产品、博客，新的Web网页
- 但是它们也会周期性地从头开始重新构建一个全新的索引
  - 查询处理将会转到新索引上去，同时将旧的索引删除

总结 — 索引构建

- 基于排序的索引构建算法
  - 它是一种最原始的在内存中进行倒排的方法
  - 基于块的排序索引算法BSBI
    - 合并排序操作对于基于磁盘的排序来说很高效(避免寻道)
- 内存式单遍扫描索引构建算法SPIMI
  - 没有全局的词典
    - 对每个块都生成单独的词典
  - 不对倒排记录进行排序
    - 有新的倒排记录出现时，直接在倒排记录表中增加一项
- 采用**MapReduce**的**分布式**索引构建算法
- 动态索引构建算法：多个索引，对数合并
- 搜索引擎：周期性索引重构

索引压缩

提纲

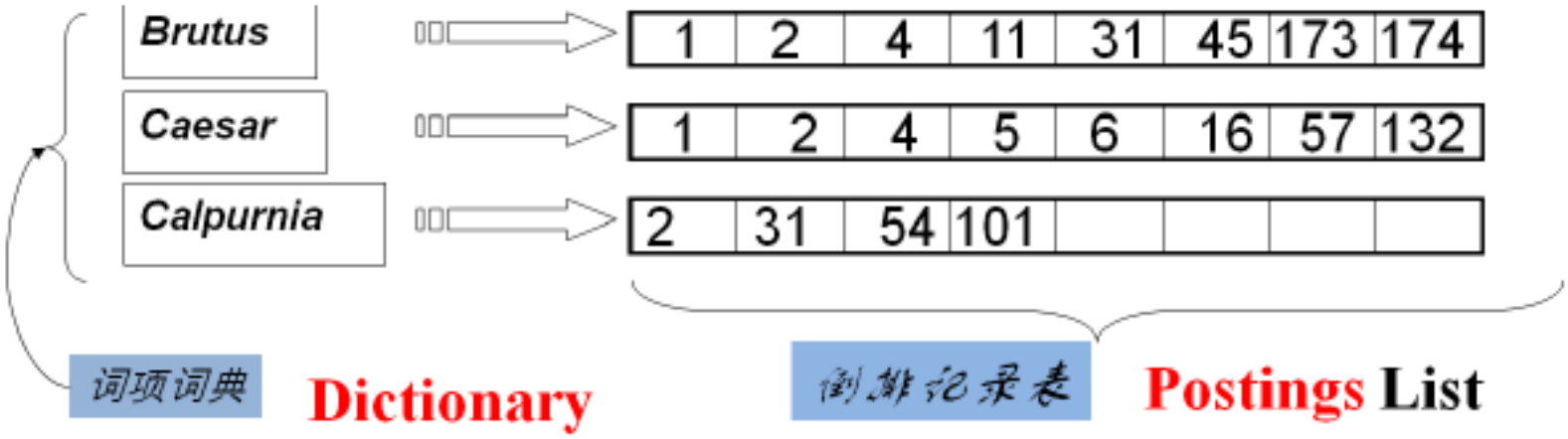
- 压缩
- 词项统计量
- 词典压缩
- 倒排记录表压缩

索引压缩

- 统计信息(对RCV1语料库)
  - 词典和倒排记录表将会有多大?
- 词典压缩
- 倒排记录表压缩

为什么要压缩

怎么压缩



为什么要压缩(一般来说)?

- 节省磁盘空间
  - 省钱
- 提高内存的利用率
  - 提高速度
- 加快数据从磁盘到内存的传输速度
  - [读取压缩数据][解压缩]比直接[读取未压缩的数据]快
  - 前提：解压缩算法要很快
    - 目前所用的解压缩算法在现代硬件上运行相当快

为什么要压缩倒排索引？

- 词典
  - 压缩的足够小以便能够放入内存中
  - 当词典足够小时，也可以在内存中存储一部分倒排记录表
- 倒排记录文件
  - 减少所需的磁盘空间
  - 减少从磁盘读取倒排记录文件所需的时间
  - 大的搜索引擎在内存中存储了很大一部分倒排记录表
    - 压缩可以在内存中存储的更多
- 将设计各种基于IR系统的压缩架构

提纲

- 压缩
- 词项统计量
- 词典压缩
- 倒排记录表压缩

回顾 Reuters-RCV1语料库

符号	含义	值
N	文档总数	800 000
L_{ave}	每篇文档的平均词条(Token)数目	200
M	词项(Term)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
T	词条(Token)总数	160 000 000

索引参数 vs. 索引内容

	不同词项			无位置信息倒排记录			词条		
	词典			无位置信息倒排表			包含位置信息的倒排表		
	数目	Δ%	T%	数目(K)	Δ%	T%	数目(K)	Δ%	T%
未过滤	484,494			109,971			197,879		
无数字	474,723	-2	-2	100,680	-8	-8	179,158	-9	-9
大小写转换	391,523	-17	-19	96,969	-3	-12	179,158	0	-9
30个停用词	391,493	0	-19	83,390	-14	-24	121,858	-31	-38
150个停用词	391,373	0	-19	67,002	-30	-39	94,517	-47	-52
词干还原	322,383	-17	-33	63,812	-4	-42	94,517	0	-52

近似是0
30个与391,523个相比

因考虑位置信息，所以不管大写，小写，都依然存在

Δ%表示和前一行相比数目的减少比率，而“停用词”均使用“大小写转换”那行作为基准。T%表示以“未过滤”为基准。
词条的数目实际上等于倒排记录表中的位置信息个数

有损(Lossy) vs. 无损(Lossless)压缩

- 无损压缩：压缩之后所有原始信息都被保留
  - 在IR系统中常采用无损压缩
- 有损压缩：丢掉一些信息
- 一些预处理步骤可以看成是有损压缩：大小写转换，停用词剔除，词干还原，数字去除
- 第7章：那些削减的倒排记录项都不太可能在查询结果的前k个列表中出现。
  - 对于前k个返回结果来说，这几乎是无损的

- 有损还是无损与需求相关！！

词汇量 vs. 文档集大小

- 词项的词汇量有多大？
  - 也就是说，有多少个不同的词？
- 可以假定一个上界吗？
  - 实际上并不可以：长度为20的不同单词至少有$70^{20}=10^{37}$个
- 实际中，词汇量会随着文档集大小的增大而增长
  - 尤其当采用Unicode编码时

- Heaps定律：$M = kT^b$
  - M是词项的数目，T是文档集中词条的个数
  - 参数k和b的典型取值为：$30 \leq k \leq 100$ 和 $b \approx 0.5$
- 词汇量大小M和文档集大小T在对数空间中，存在着斜率为½的线性关系
  - 在对数空间中，这是这两者之间存在的最简单的关系
  - 这是一个经验发现(“empirical law”)

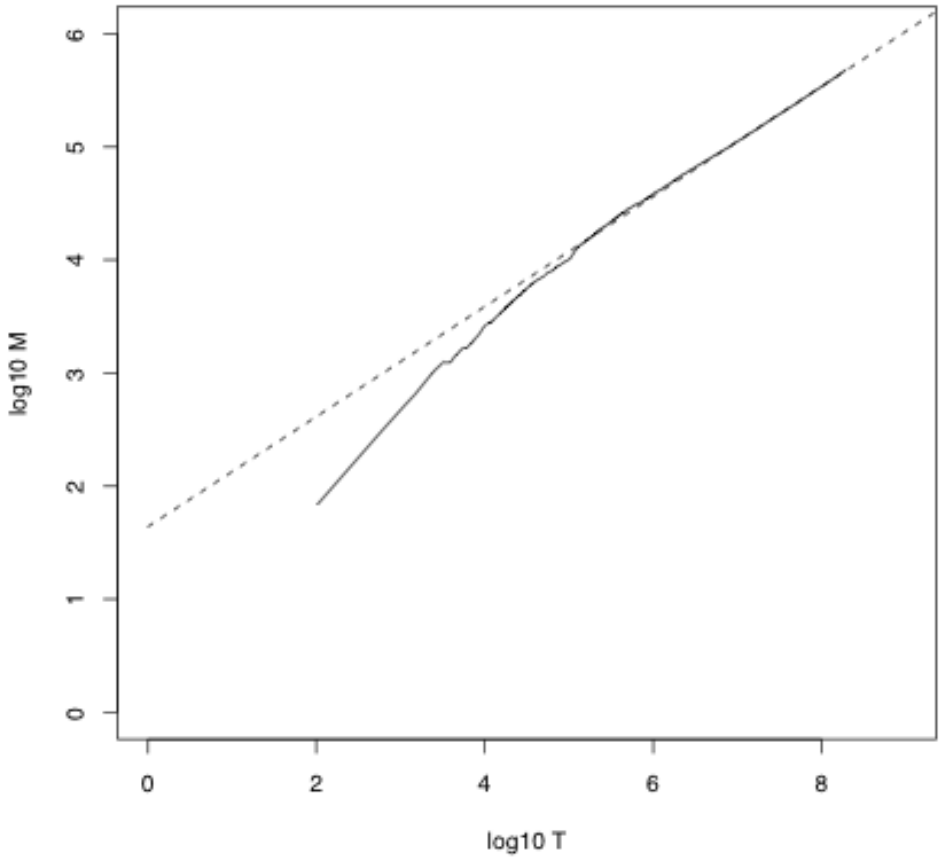
Heaps定律是Heaps在1978年一本关于信息挖掘的专著中提出的。事实上，他观察到在语言系统中，不同单词的数目与文本篇幅（所有出现的单词累积数目）之间存在幂函数的关系，其幂指数小于1。

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



M是**词项**的数目，T是文档集中**词条**的个数

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

Zipf定律

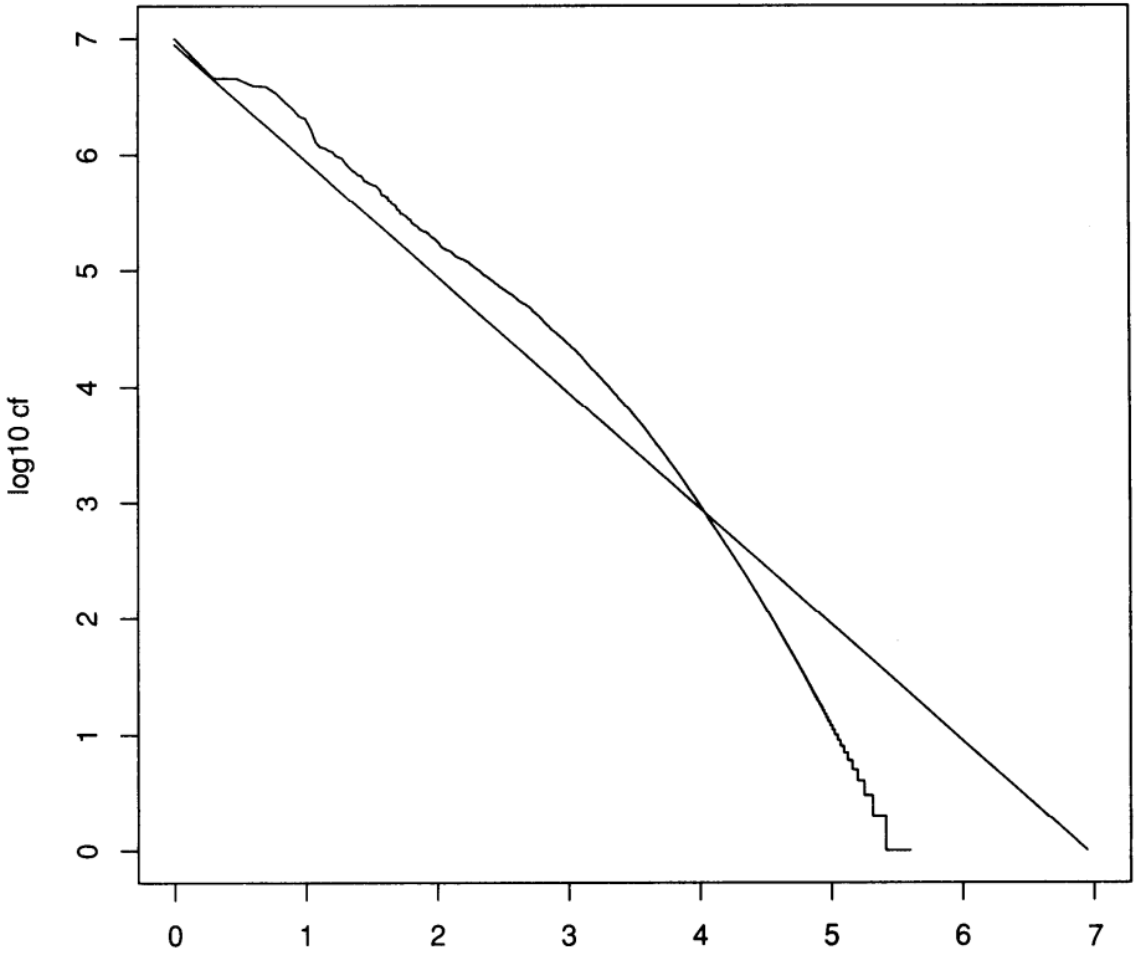
- Heaps定律提供了对文档集中词汇量的估计
- 我们还想了解词项在文档中的分布情况
- 在自然语言中，只有很少一些非常高频的词项，而其它绝大部分都是很生僻的词项。
- **Zipf定律**：排名第i多的词项的文档集频率与$1/i$ 成正比
  - $cf_i \propto \frac{1}{i} = \frac{K}{i}$，$K$是一个归一化常数
  - cf_i是文档集频率：词项t_i在文档集中出现的次数
- Zipf定律是Zipf在1949年的一本关于人类定位的最小作用原理的书中首先提出的，最令人难忘的例子是在人类语言中，如果以单词出现的频次将所有单词排序，用横坐标表示序号，纵坐标表示对应的频次，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。

Zipf定律推论

- 如果最高频的词项(the)出现了cf_1次
- 那么第2高频的词项(of)出现了$cf_1/2$次
- 第3高频的词项(and)出现了$cf_1/3$次
- 等价的：$cf_i = K/i$ 中K是归一化因子，所以
  - $\log cf_i = \log K - \log i$
  - $\log cf_i$和$\log i$之间存在着线性关系
- 另一个幂定律关系

Reuters-RCV1文档集上的Zipf定律

- 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

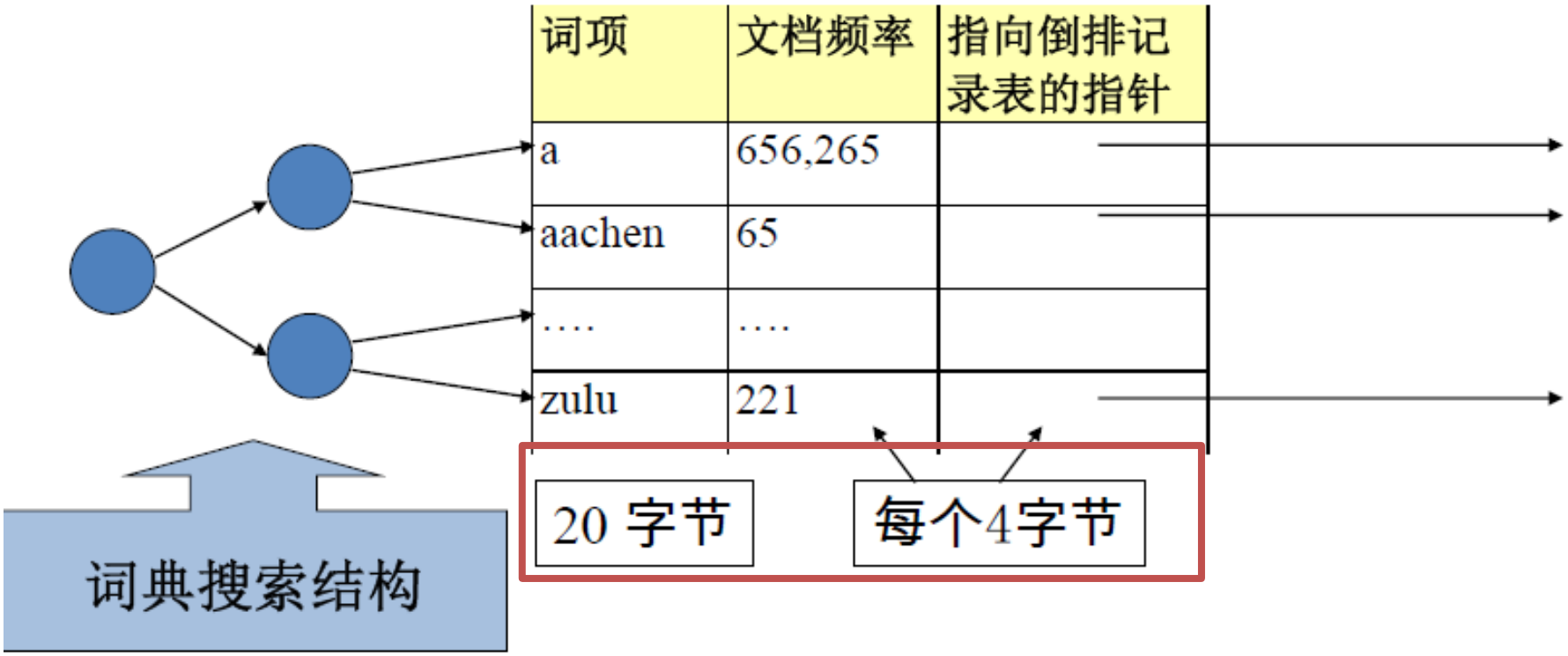
- 压缩
- 词项统计量
- 词典压缩
- 倒排记录表压缩

为什么要压缩词典？

- 搜索从词典开始
- 需要将词典放入内存中
- 和其他应用程序共享内存资源
- 手机或者嵌入式设备通常只有很小的内存
- 即使词典不存入内存中，也希望它能比较小，以便搜索能快速启动
- 所以，压缩词典非常重要

词典存储

- 定长数组存储
- 400,000词项×28B/词项 = 11.2 MB

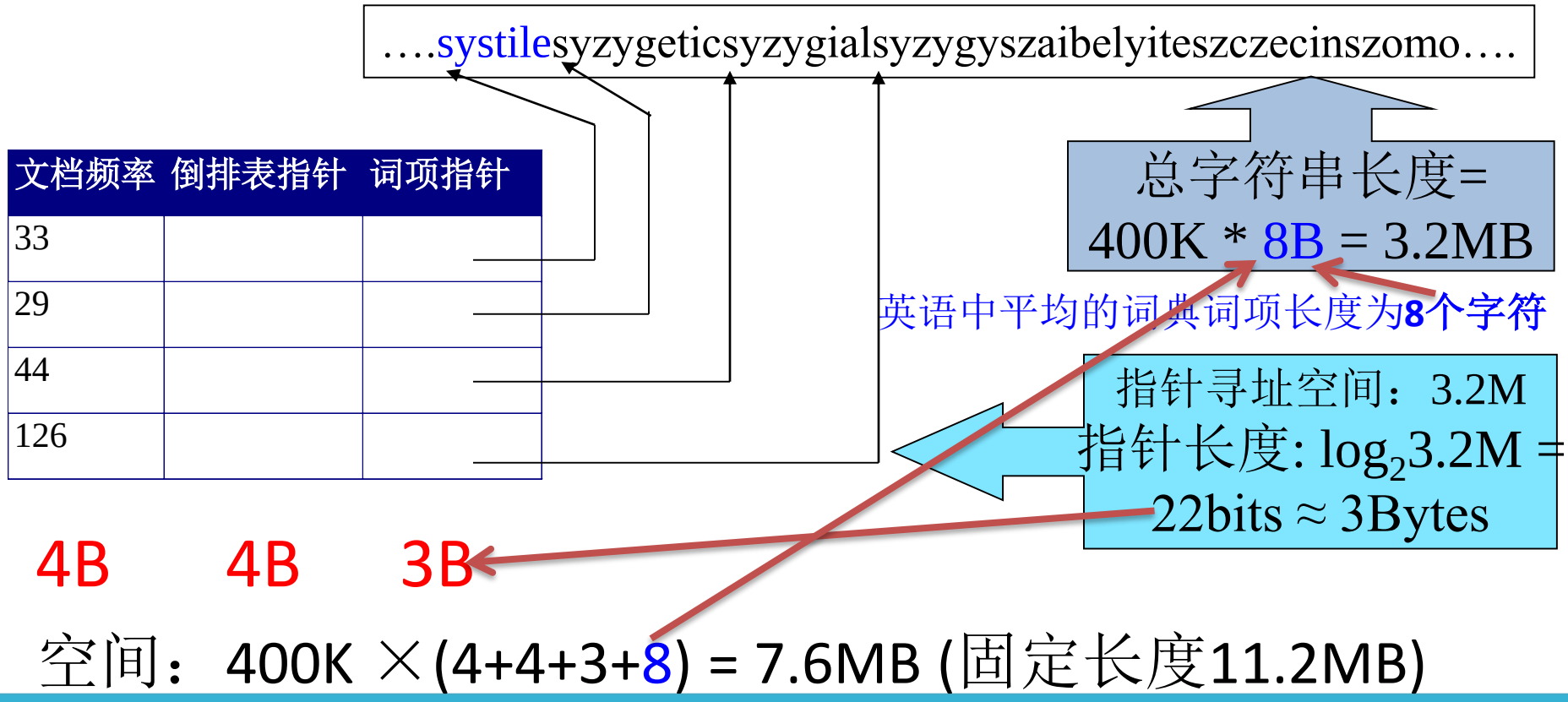


定长方法存储词项浪费空间

- 词项那一列大部分的字节都被浪费 —为每个词项分配了20字节的固定长度。
  - 但仍然不能解决“supercalifragilisticexpialidocious”和“hydrochlorofluorocarbons”
- 书面英文中单词的平均长度约为4.5个字符
- 英语中平均的词典词项长度为**8个字符**
  - 平均会有12个字符的空间浪费
- 较短的词项支配了词条的数目但是并不是典型的平均值，即较短的词项占绝大多数

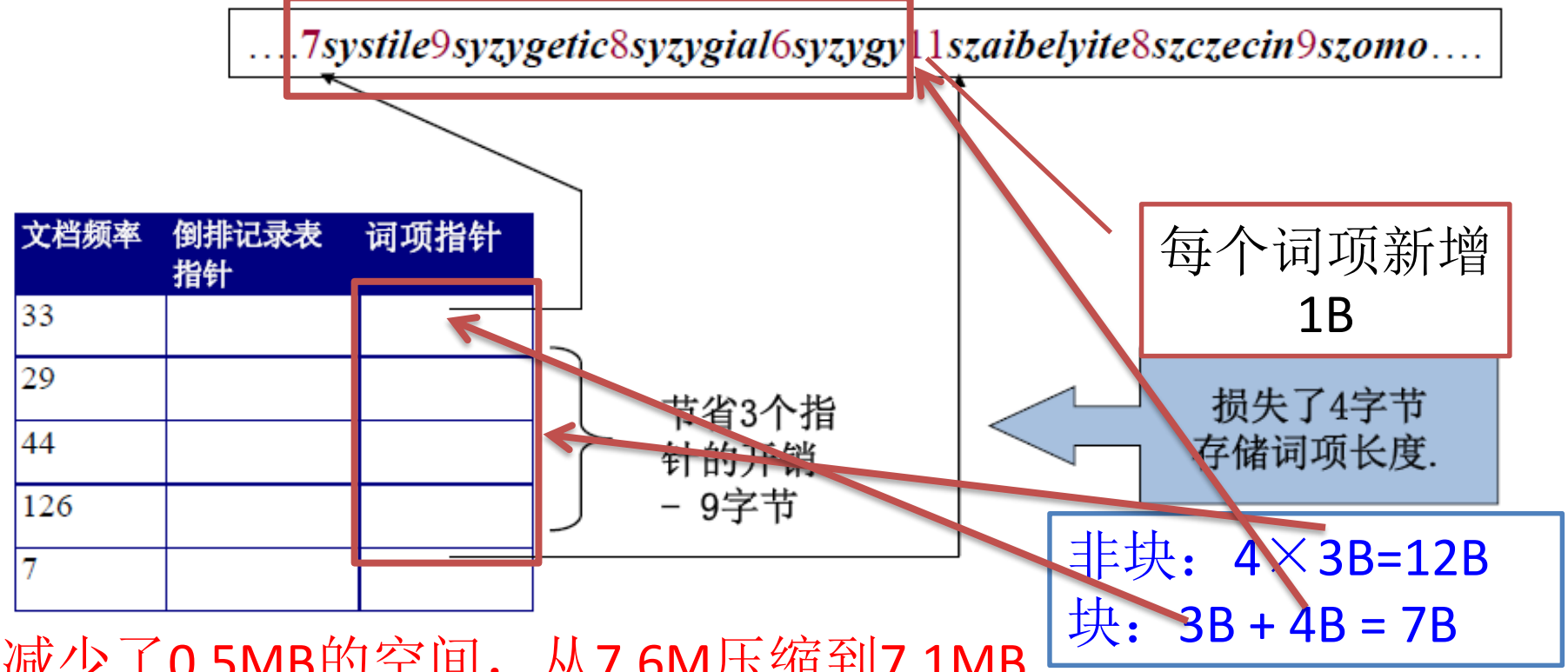
压缩词项列表：将词典看成单一字符串（Dictionary-as-a-String）

- 将所有词项存储为一个长字符串：
  - 指向下一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



按块存储（Blocking）

- 每k个词项分成一块，只保留第一个指针
  - 下面的例子：k=4
- 需要存储词项长度(额外1字节)

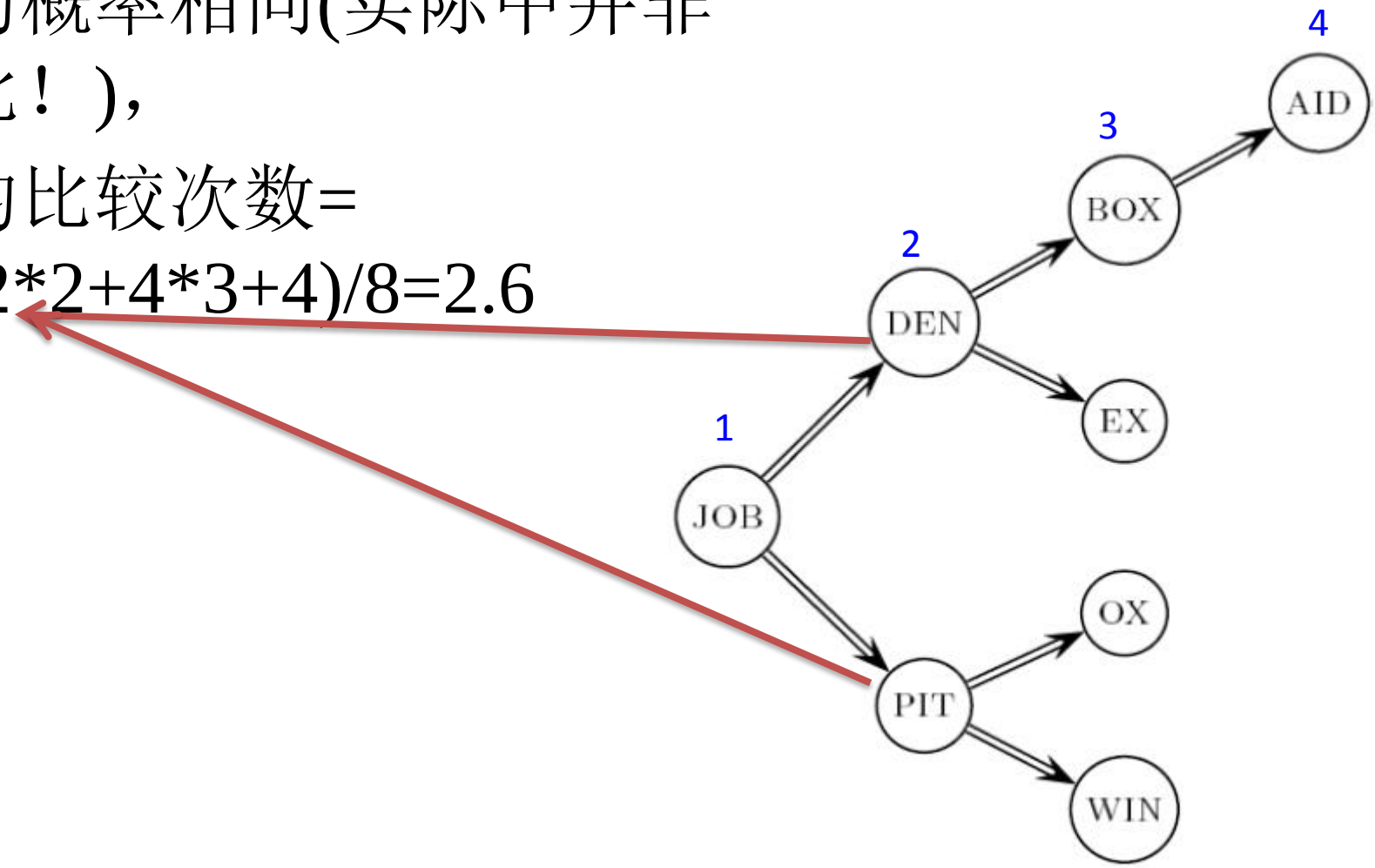


减少了0.5MB的空间，从7.6M压缩到7.1MB

讨论：k取多少 合适？大？小？

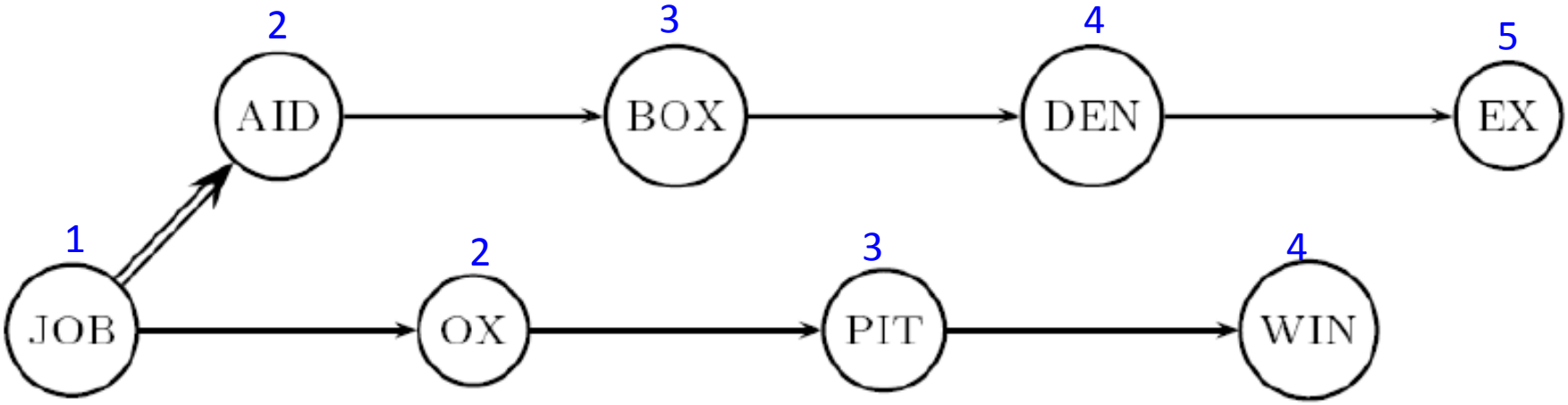
未压缩词典的搜索

- 假设词典中每个词项被查询的概率相同(实际中并非如此！)，
- 平均比较次数=(1+2*2+4*3+4)/8=2.6



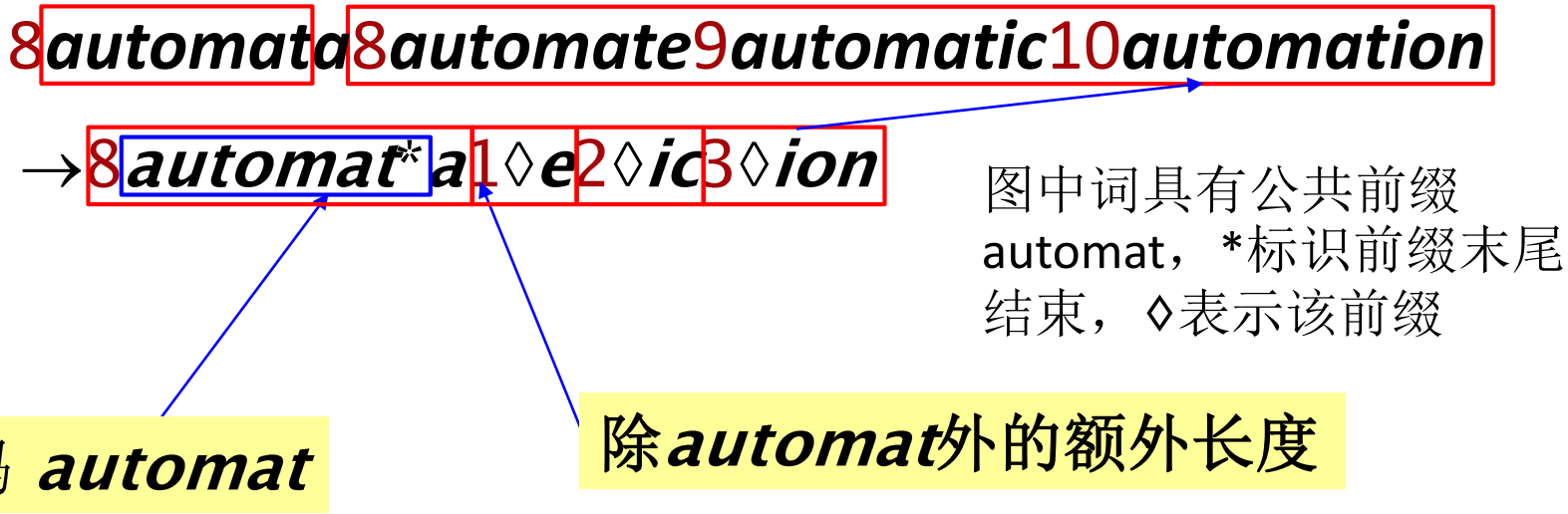
按块存储方式下的词典搜索

- 二分查找只能在块外进行
  - 然后在块内进行线性查找(串行查找)得到最后的词项位置
- 块大小为4(二分树)，平均比较次数
  =(1+2*2+2*3+2*4+5)/8 = 3



前端编码（Front coding）

- 前端编码
  - 按照词典顺序排列的连续词项之间往往具有公共前缀
  - (块内k个词项的最后k-1个)



小结 词典压缩：RCV1文档集的词典压缩结果

数据结构	压缩后大小(MB)
定长数组	11.2
长字符串+词项指针	7.6
按块存储，k=4	7.1
按块存储+前端编码	5.9

提纲

- 压缩
- 词项统计量
- 词典压缩
- 倒排记录表压缩

倒排记录表压缩

- 倒排记录表远大于词典，至少为10倍
- 迫切要求：紧密地存储每一个倒排记录表
- 每个倒排记录用文档ID来定义
- 对Reuters (800,000文档)来说，当使用4字节(定长)整数表示时，每个文档ID需要32bit
- 或者，可以用$\log_2 800{,}000 \approx 20$ bits 来表示每个文档ID
- 目标：用远小于20bit来表示每个文档ID

100M(倒排记录数目) * 20b / 8 (b/B) = 250MB

预处理前后词项、词条数目

	不同词项			无位置信息倒排记录			词条		
	词典			无位置信息倒排表			包含位置信息的倒排表		
	数目	Δ%	T%	数目(K)	Δ%	T%	数目(K)	Δ%	T%
未过滤	484,494			109,971			197,879		
无数字	474,723	-2	-2	100,680	-8	-8	179,158	-9	-9
大小写转换	391,523	-17	-19	96,969	-3	-12	179,158	0	-9
30个停用词	391,493	0	-19	83,390	-14	-24	121,858	-31	-38
150个停用词	391,373	0	-19	67,002	-30	-39	94,517	-47	-52
词干还原	322,383	-17	-33	63,812	-4	-42	94,517	0	-52

近似是0
30个与391,523个相比

因考虑位置信息，所以不管大写，小写，都依然存在

Δ%表示和前一行相比数目的减少比率，而“停用词”均使用“大小写转换”那行作为基准。T%表示以“未过滤”为基准。
词条实际上表示了考虑位置信息。

倒排记录表：相反的两点

- 像“arachnocentric”这样的词项可能在一百万个文档中才会出现一次 – 可以用$\log_2 1M \approx 20$ bits来存储这一倒排记录。
- 像“the”这样的词项在每个文档中都会出现，所以对它采用20bit/倒排记录太浪费了
  - 这种情况更希望是0/1的bit向量

规律的探寻：倒排记录表项中文档ID的间距(GAP)

- 按照文档ID的递增顺序来存储一个词项的倒排列表。
  - Computer：33, 47, 154, 159, 202, …
- 结论： 可以存储间距
  - 33, 14, 107, 5, 43, …
- 期望：绝大多数间距存储空间都远小于20bit

找找GAP：3个倒排记录表项

表5-3　对文档ID的间距而不是文档ID进行编码

	编码对象	倒排记录表					
the	文档ID	…		283042	283043	283044	283045	…
	文档ID间距				1	1	2	…
computer	文档ID	…		283047	283154	283159	283202	…
	文档ID间距				107	5	43	…
arachnocentric	文档ID	252000		500100				
	文档ID间距	252000	248100					

注：比如，对于 computer，存储间距序列 107, 5, 43, . . . ,而不是文档 ID 序列 283154, 283159, 283202,…。

当然，第一个文档 ID 仍然被保留（表中仅显示了arachnocentric的第一个文档ID）。

可变长度编码

- 目标：
  - 对于arachnocentric(低频), 使用20bit/间距项
  - 对于the(高频)，使用1 bit/间距项
- 如果词项的平均间距为G，我们想使用$\log_2 G$ bit/间距项
- 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 ：对一些小数字使用短码来实现

例子

214577= 0001101 0001100 0110001

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

824 5 214577

关键特性：VB编码过的倒排记录表是唯一前缀可解的

对一个小的间距(5)，VB编码使用了一整个字节，浪费存储空间

GAP→可变字节码(Variable Byte)

- 对一个间距值G，想用最少的所需字节来表示$\log_2 G$ bit
- 先用一个字节来存储G，并分配1bit作为延续位c
- 如果 $G \leq 127$，对7位有效码采用二进制编码并设延续位c=1 (表示结束)
- 若$G > 127$，则先对G低阶的7位编码，然后采取相同的算法用额外的字节对高阶bit位进行编码
- 设置最后一个字节的延续位为1(c=1)，其他字节的c=0 (表示未结束)

其它的可变单位编码

- VB编码思想也可应用在与字节不同的单位上：32bit(words)，16bit，4bit(nibble)
- 可变字节编码在那些很小的间距上浪费了空间 — 半字节在这种情况下表现得更好
- 可变字节编码
  - 被很多商业/研究系统所使用
  - 实现简单，能够在时间和空间之间达到一个非常好的平衡点

RCV1压缩

表5-6　Reuters-RCV1中的索引及词典压缩

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（32bit位字）	400.0
倒排记录表，未压缩（20bit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

- 现在可以为布尔查询创建一个索引，即高效又非常节省空间
- 只有文档集总大小的4%
- 在文档集中只有文本总大小的10-15%
- 但是，忽略了索引的位置信息
- 因此，在实际中，索引所节省的空间并没有这么多