信息检索导论

An Introduction to Information Retrieval

第4讲 索引构建

Index construction

授课人：李波

中国科学院信息工程研究所/国科大网络空间安全学院

*改编自"An introduction to Information retrieval"网上公开的课件，地址 http://nlp.stanford.edu/IR-book/

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

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

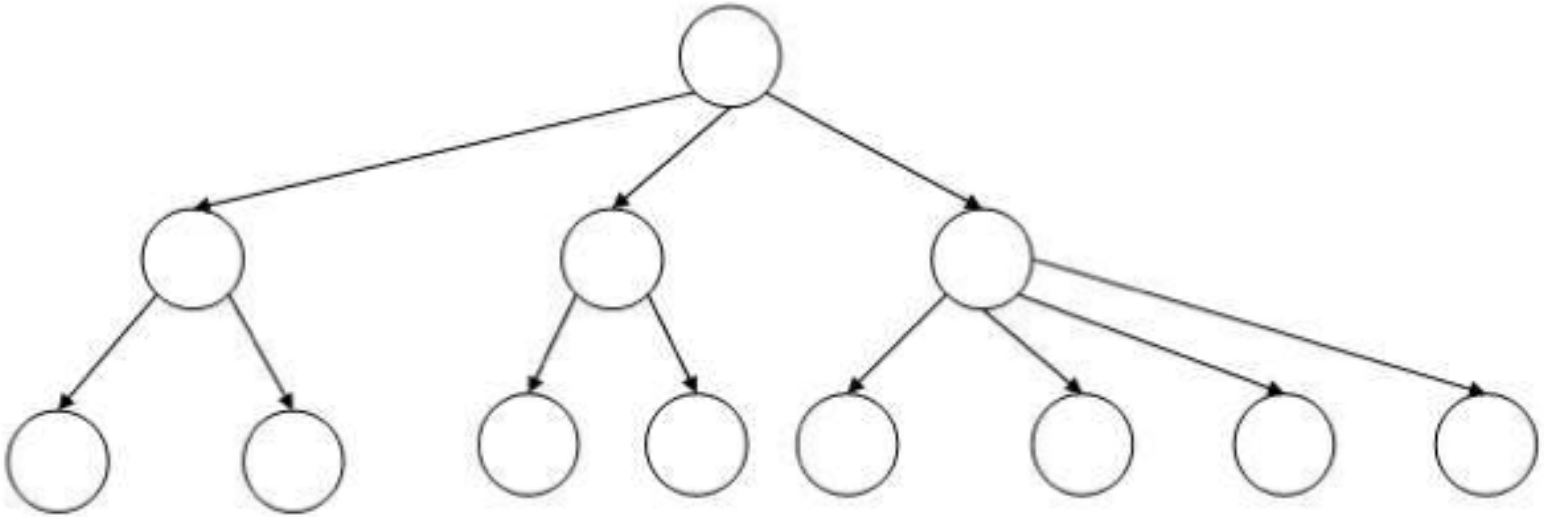
- 哈希表：
  - 定位速度快，常数时间
  - 不易支持动态变化的词典
  - 不支持前缀查询
- 树结构：二叉树、B-树等等
  - 定位速度为对数时间
  - 二叉(平衡)树支持动态变化，但是重排代价大。B-树能够缓解该问题
  - 支持前缀查询

哈希表(散列表)

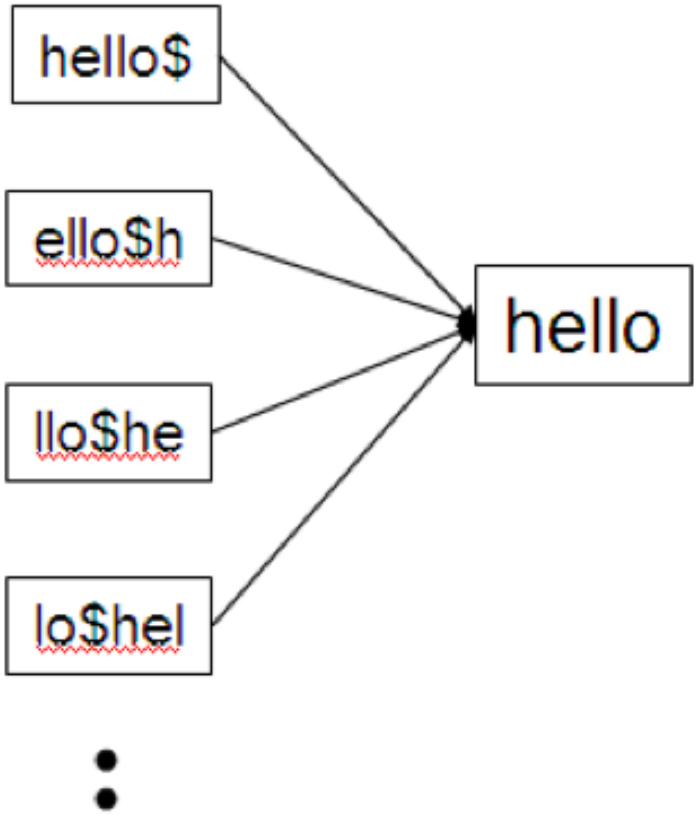
哈希函数，输入词项，输出正整数(通常是地址)

f(信息)=18, f(数据)=19,
f(挖掘)=19

B-树



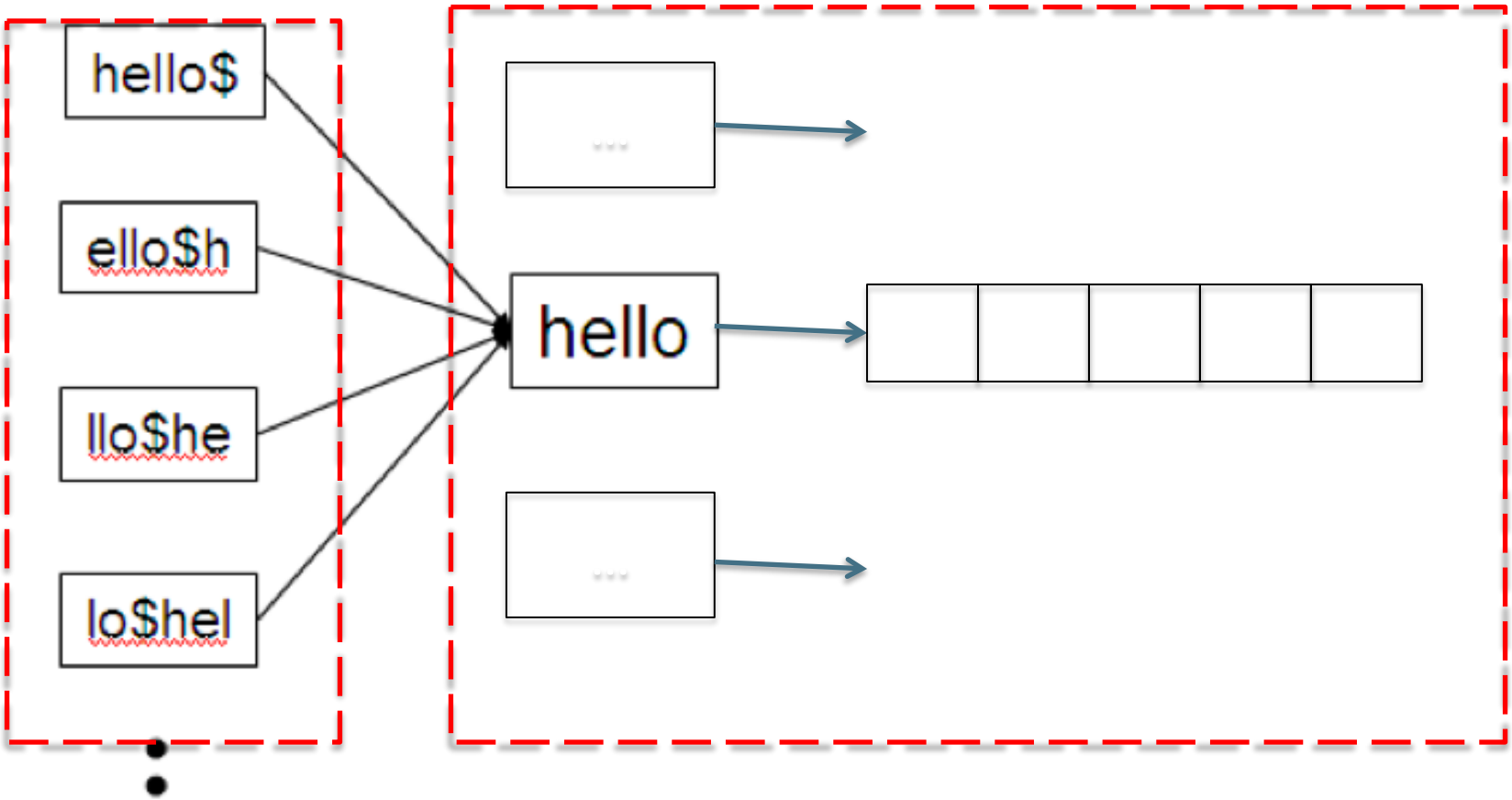
通配查询处理：轮排索引



查询:

- 对 X, 在轮排索引中查找X$字符串
- 对X*, 查找以$X为前缀的字符串
- 对*X, 查找以X$为前缀的字符串
- 对*X*, 查找以X为前缀的字符串
- 对 X*Y, 查找以Y$X为前缀的字符串

轮排索引示意图



轮排索引 (通配查询→词项，采用B树来组织)

传统倒排索引 (词项→文档)

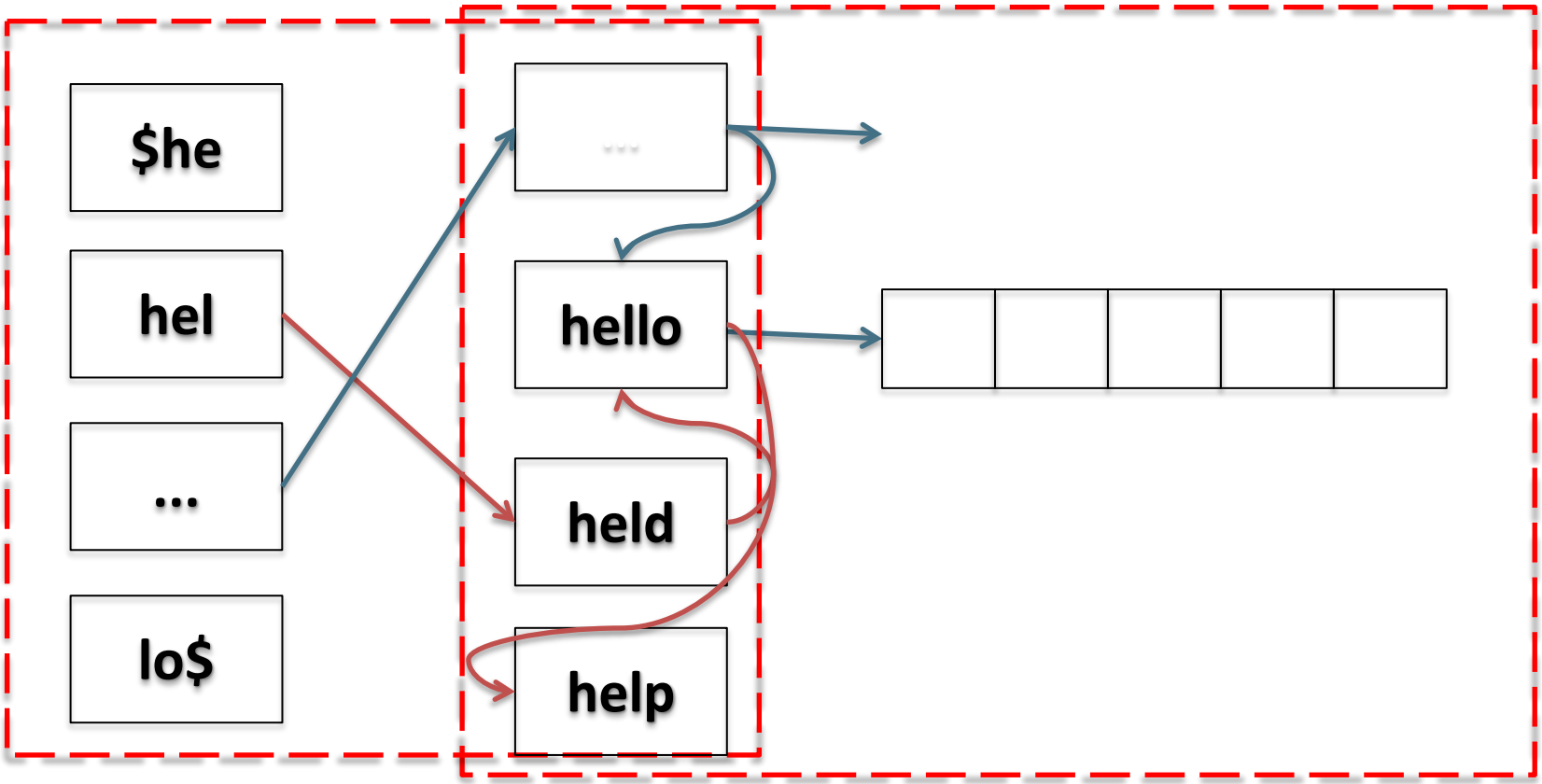
轮排索引使用过程

- 通配查询加上$号后，将通配符移到右部
  - hel*o --> hel*o$ -->o$hel*
- 在轮排索引中查找相应字符串得到相应词项集合
  - 查找前缀为o$hel字符串的词项集合
- 将词项对应的文档ID列出
  - 查词项-文档倒排索引，将上述词项集合对应的文档ID取出

通配查询处理：k-gram索引

- 枚举一个词项中所有连续的k个字符构成k-gram 。
  - 2-gram称为二元组(bigram)
- 例子: April is the cruelest month: $a ap pr ri il l$ $i is s$ $t th he e$ $c cr ru ue el le es st t$ $m mo on nt h$
  - 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)

k-gram索引示意图



k-gram索引 (k-gram→词项，本质上是一种倒排索引)

传统倒排索引 (词项→文档)

k-gram索引使用过程

- 通配查询加上$号后，将查询转换成布尔查询
  - hel*o --> $hel*o$ --> $h AND he AND el AND o$
- 在k-gram索引中查找满足上述布尔表达式的词项集合
  - 分别找出$h、he、el和o$对应的词项集合，求交集
- 进行后过滤
  - 将上述词项集合中的每个词项和原始查询进行匹配，将不满足原始查询的词项去掉
- 将上述词项集合对应的文档ID取出

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：首字母相同；保证两者之间具有较长公共子串)

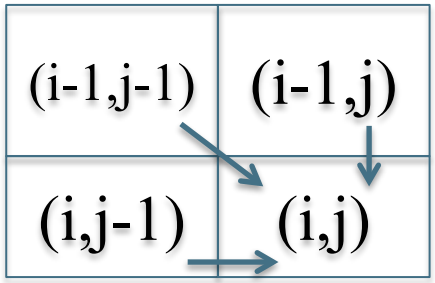
Levenshtein编辑距离计算

Copy Replace	delete
insert	*MIN*

LEVENSHTEINDISTANCE(s_1, s_2)

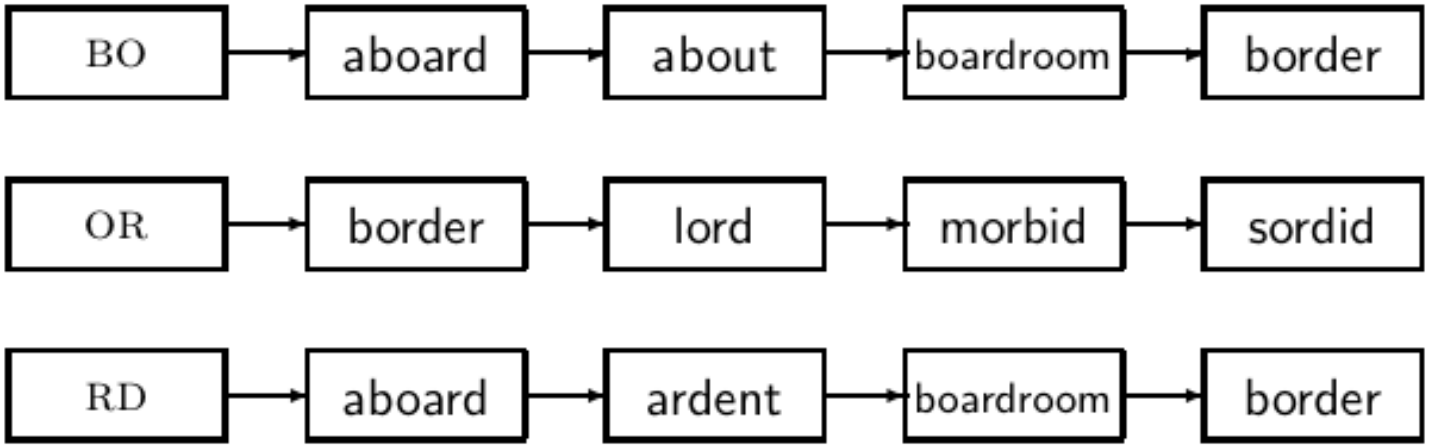
```
 1  for i ← 0 to |s1|
 2  do m[i, 0] = i
 3  for j ← 0 to |s2|
 4  do m[0, j] = j
 5  for i ← 1 to |s1|
 6  do for j ← 1 to |s2|
 7       do if s1[i] = s2[j]
 8            then m[i, j] = min{m[i − 1, j] + 1, m[i, j − 1] + 1, m[i − 1, j − 1]}
 9            else m[i, j] = min{m[i − 1, j] + 1, m[i,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

Operations: insert, delete, replace, copy



基于k-gram索引的拼写校正

查询“bord”的2-gram索引：



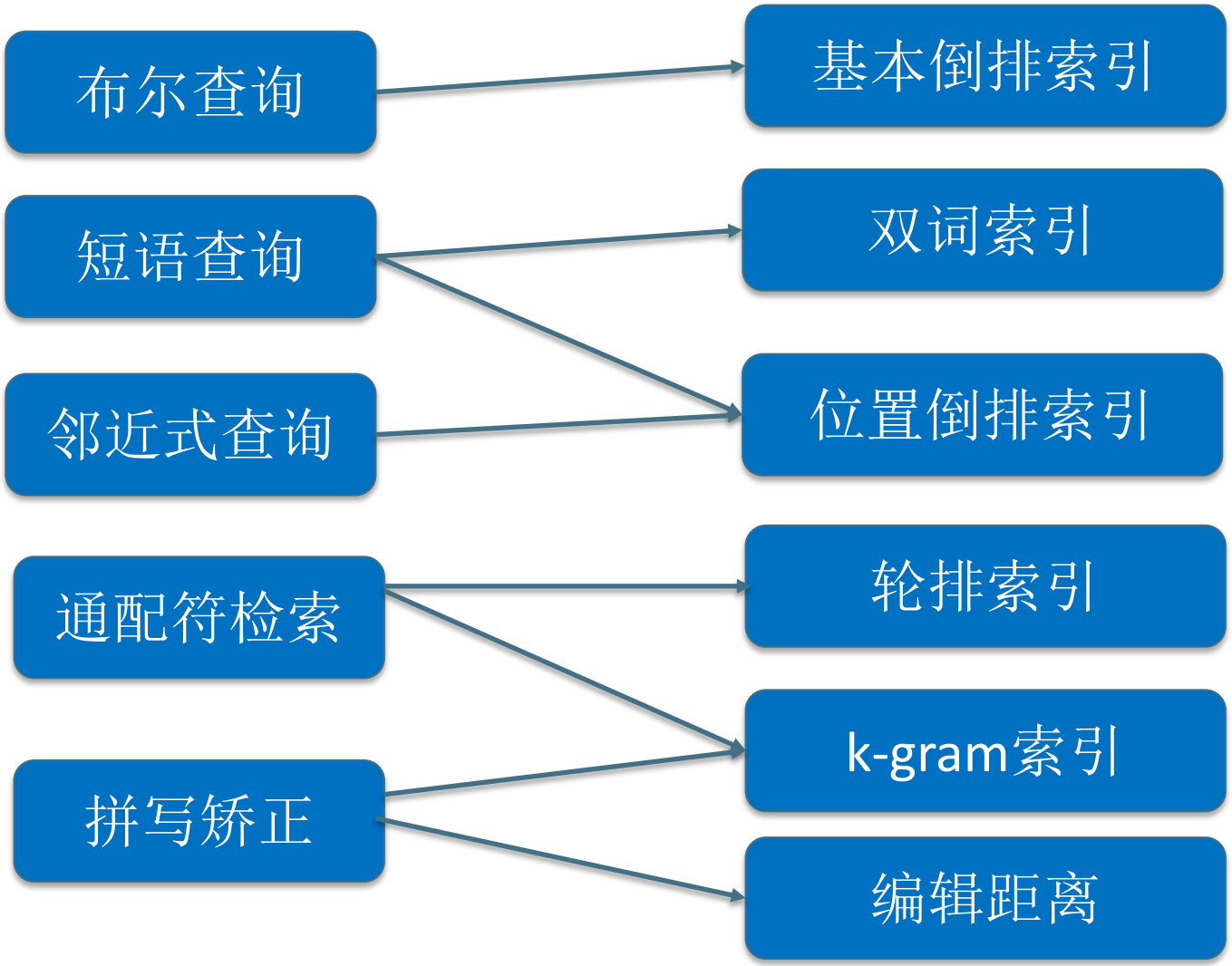
命中至少两个2-gram的词项：aboard、boardroom、border

$$J(A,B)=|A\cap B|/|A\cup B|$$

q=bord, t=boardroom

J=2/8+3−2

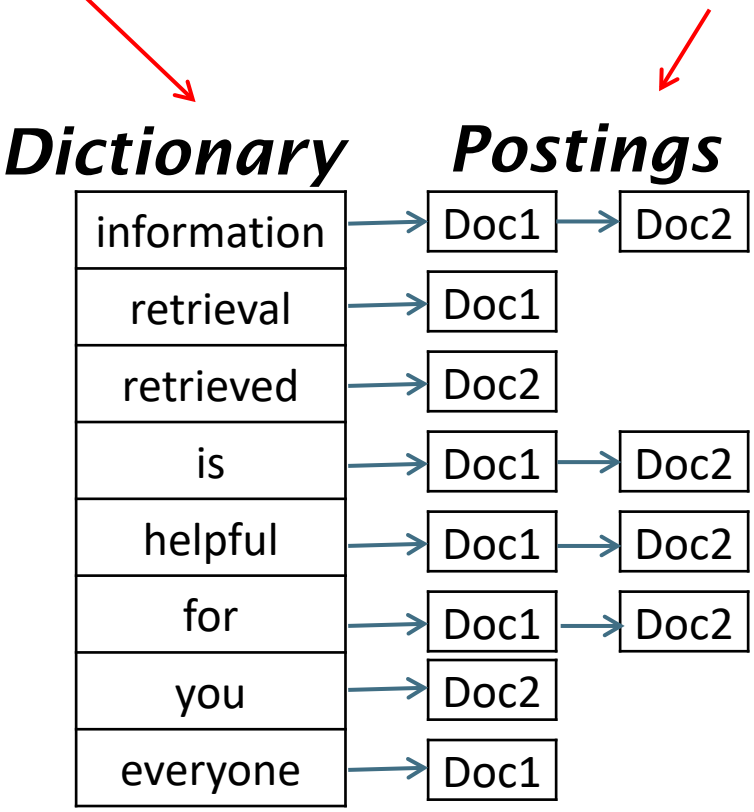
倒排索引技术小结


布尔查询
短语查询
邻近式查询
通配符检索
拼写矫正
基本倒排索引
双词索引
位置倒排索引
轮排索引
k-gram索引
编辑距离

倒排索引技术小结

- 容错式检索（lec3）：
e.g., 通配符查询，拼写校正

- 跳表、临近检索（lec2）：
e.g., 短语查询



本讲内容

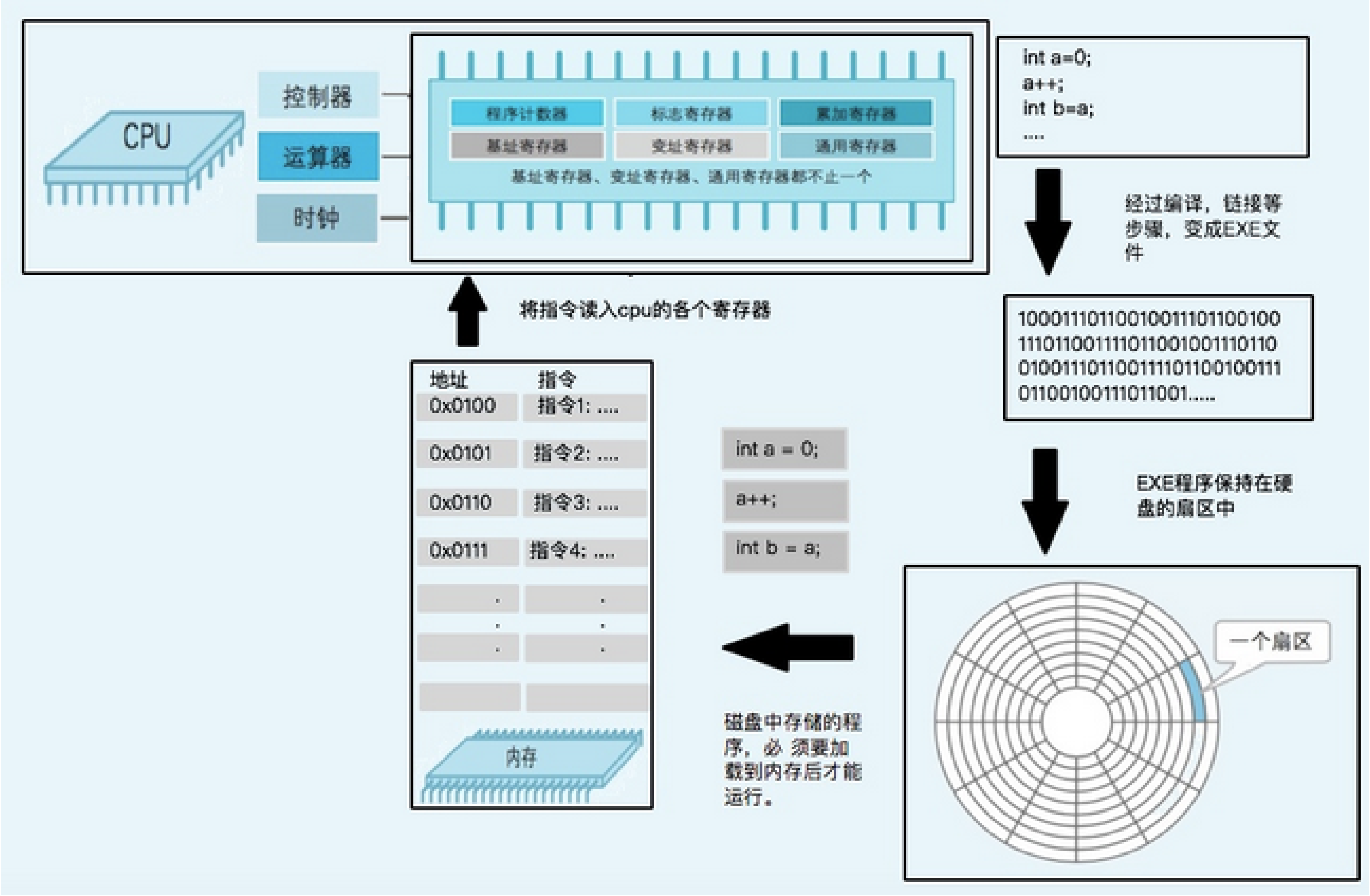
- 两种索引构建算法: BSBI (简单) 和 SPIMI (更符合实际情况)
- 分布式索引构建: MapReduce
- 动态索引构建: 如何随着文档集变化更新索引

提纲

硬件基础知识(1)

- 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(2)


CPU
控制器
运算器
时钟
程序计数器
标志寄存器
累加寄存器
基址寄存器
变址寄存器
通用寄存器
基址寄存器、变址寄存器、通用寄存器都不止一个
int a=0;
a++;
int b=a;
....
经过编译，链接等步骤，变成EXE文件
1000111011001001110110010011101100111101100100111011001001110111001111011001001110110010011101100.....
EXE程序保持在硬盘的扇区中
一个扇区
磁盘中存储的程序，必 须要加载到内存后才能运行。
int a = 0;
a++;
int b = a;
将指令读入cpu的各个寄存器
地址
指令
0x0100
指令1: ....
0x0101
指令2: ....
0x0110
指令3: ....
0x0111
指令4: ....
内存

硬件基础知识(3)

- 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GB（10~100GB）级别的内存，TB级的硬盘
- 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

一些统计数据(2008年)

符号	含义	值
s	平均寻道时间	5 ms = 5×10^{-3} s
b	每个字节的传输时间	0.02 μs = 2×10^{-8} s
	处理器时钟频率	10^{9} s^{-1}
p	底层操作时间 (e.g., 如word的比较和交换)	0.01 μs = 10^{-8} s
	内存大小	几GB
	磁盘大小	1 TB或更多

Reuters RCV1 语料库

- 《莎士比亚全集》规模较小（10MB），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

Reuters RCV1语料库的统计信息

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

课堂练习:

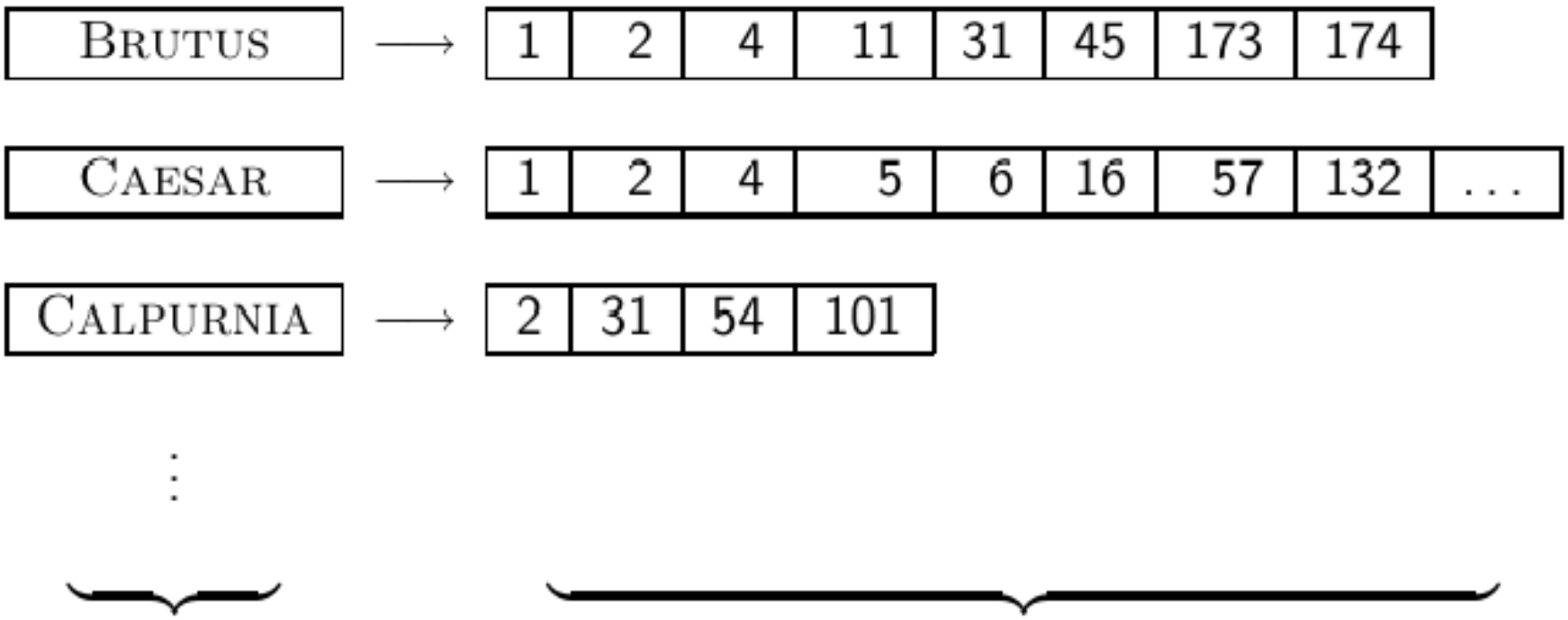
(1) 一个词项的平均出现次数是多少？即一个词项平均对应几个词条？

(2) 每个词项的平均倒排记录数目是多少？

(3) 每个词条字节数为4.5 vs. 每个词项平均字节数 7.5，为什么有这样的区别？

(4) 带位置信息索引的倒排记录数目是多少？

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

倒排索引的结构

- 字典: 空间不算太大
  - 快速随机访问性能要求高
  - 需常驻内存
    - 哈希表/B-树/trie
- 倒排表：空间巨大
  - 存储在磁盘
  - 通常是顺序访问
  - 包含docID、词频、位置信息等
  - 需要进行压缩

"Key data structure underlying modern IR"
- Christopher D. Manning

提纲

第一讲中介绍的基于排序的索引构建方法：在内存中对倒排记录表进行排序

- 第一步：对每篇文档，解析文档内容，抽取单词，生成单词-文档ID对

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	Doc #
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

第一讲中介绍的基于排序的索引构建方法：在内存中对倒排记录表进行排序

- 第二步：所有文档都解析完成后，按照词项进行排序，生成倒排索引

重点在该步的排序操作：
需要对100M个项进行排序！

Term	Doc #
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	Doc #
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？

Reuters RCV1索引创建需要多少时间？

- N=100,000,000个记录
- 排序比较次数：$N\log_2(N) = 2,657,542,475.91$
- 每次比较操作需要进行两次磁盘seek操作（5ms），共需：13,287,712.38 x 2 秒
  - = 26,575,424.76 seconds
  - = 442,923.75 minutes
  - = 7,382.06 hours
  - = 307.59 days
  - = 84% of a year
  - = 1% of your life

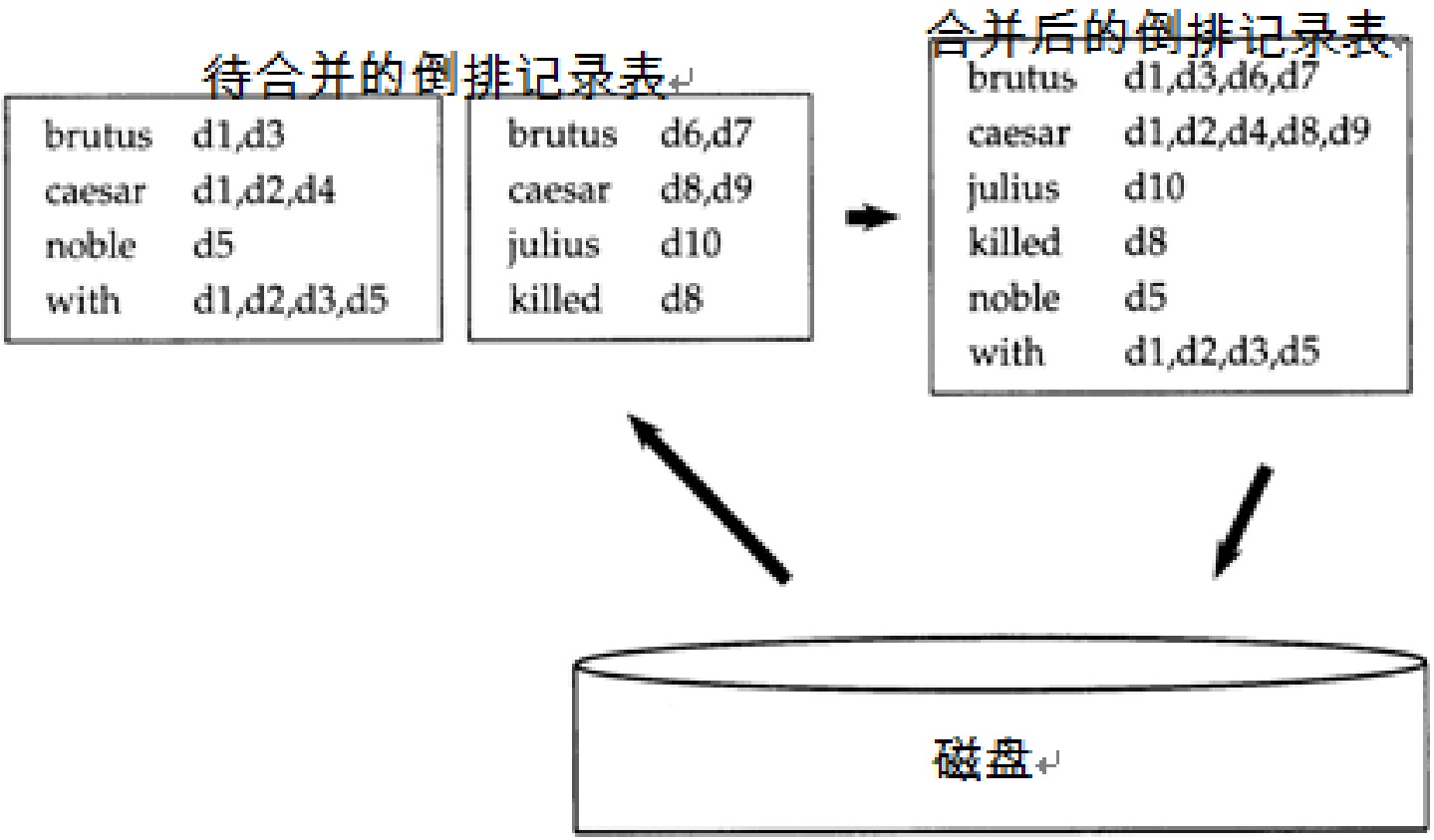
是否在磁盘上采用同样的算法？

- 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

基于排序的分块索引构建算法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部分就是确定块的大小

BSBI算法分析

- 时间复杂度是O(TlogT)
  - T是词项-文档ID对的个数
- RCV1的索引构建时间是多少？
  - 分成10个数据块，每个块的N' = 10,000,000
  - 比较次数：10* N'log2N'=2325349666.4
  - 每次比较开销(10e-8 sec) =23秒

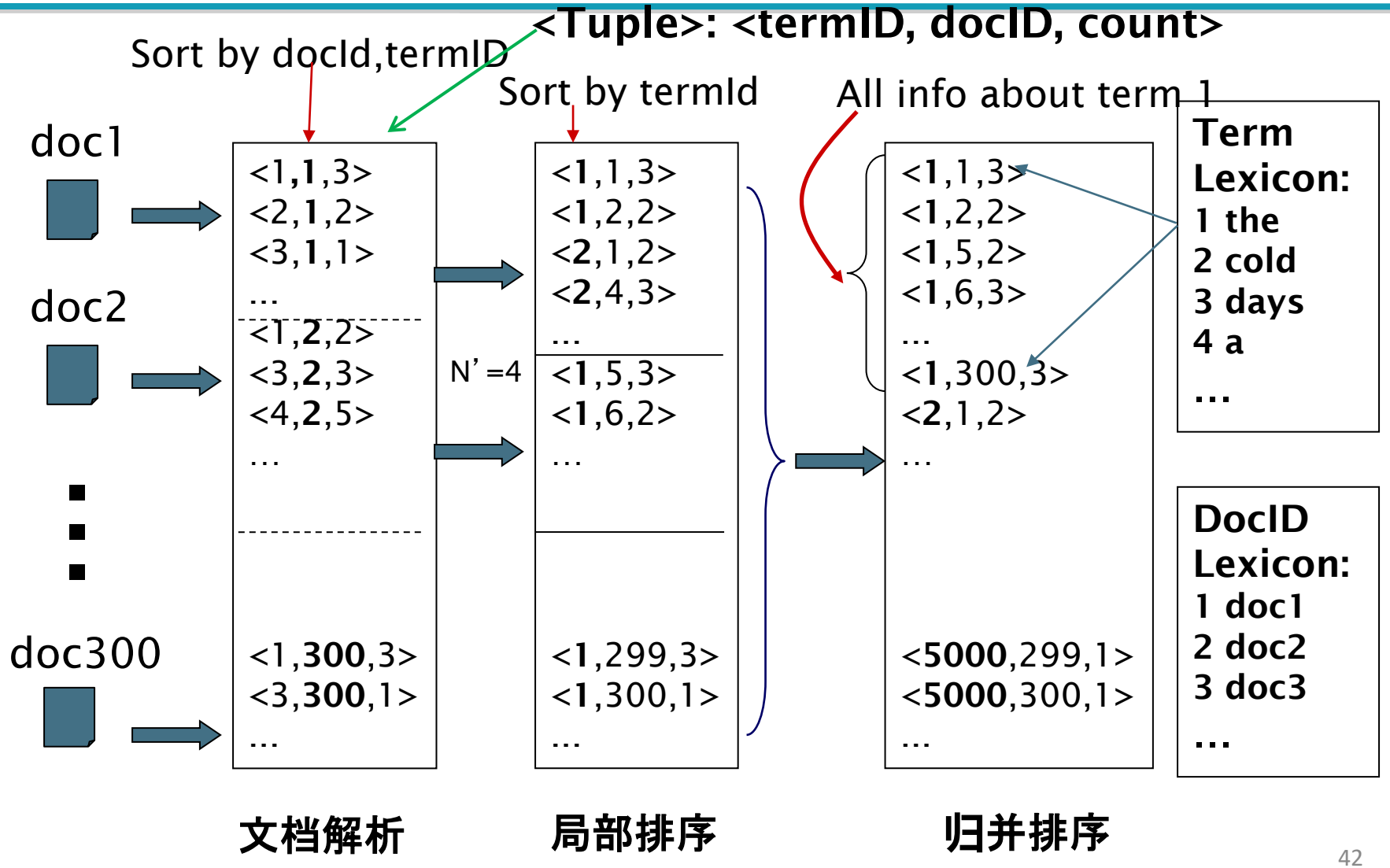
- 实际上，ParseNextBlock占用了大部分时间
- 然后是MergingBlocks
- 同样是因为磁盘寻道延迟 vs. 内存访问延迟

BSBI算法分析

```
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)
```

	step	time
1	reading of collection (line 4)	
2	10 initial sorts of 10^7 records each (line 5)	
3	writing of 10 blocks (line 6)	
4	total disk transfer time for merging (line 7)	
5	time of actual merging (line 7)	
	total	

BSBI索引构建过程总结



BSBI索引构建过程总结

- 挑战
  - 文档大小超过内存大小
- 关键步骤
  - 局部排序: 按termID排序
  - 全局归并排序
    - 同时保持docID顺序：为了后续倒排表的合并

*该方法可以实现在单节点上构建大规模语料索引！同样适用**MapReduce!***

提纲

基于排序的分块索引构建算法的问题

- BSBI很简单，但存在以下问题
  - 通常需要专门的数据结构将term映射成termID
  - 要求该映射全部在内存中放下
  - 对于大规模文档集来说，词项集合是动态变化的，实际大小也难以保证可以全部存在内存中
  - 此外，文档解析过程，需要缓存大量（TermID，DodcID）中间元素
  - 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
  - 不需要在块之间共享全局term-termID的映射
- 关键思想2: 按照词项出现的先后顺序分别构建索引
  - 中间过程无需按term/termID排序

- 基于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成一个大索引

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与BSBI的对比

- SPIMI直接将倒排记录加到倒排表中
- BSBI首先收集（TermID，DodcID）元素对
  - 然后按TermID排序
  - 然后再合并到倒排表中

- SPIMI的倒排表是动态变化的，所以中间过程无需按term进行排序
- 每个term只存储一次，更节省内存
- 复杂度为：O（T）

SPIMI与BSBI的对比

- 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

课堂练习:计算1台机器下采用BSBI方法对Google级规模数据构建索引的时间

```
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)
```

	step	time
1	reading of collection (line 4)	>100天
2	10 initial sorts of 10^7 records each (line 5)	？
3	writing of 10 blocks (line 6)	>50天
4	total disk transfer time for merging (line 7)	>50天*2
5	time of actual merging (line 7)	？
	total	

symbol	statistic	value
s	average seek time	5 ms $= 5 \times 10^{-3}$ s
b	transfer time per byte	0.02 μs $= 2 \times 10^{-8}$ s
	processor's clock rate	10^9 s^{-1}
p	lowlevel operation	0.01 μs $= 10^{-8}$ s
	number of machines	1
	size of main memory	8 GB
	size of disk space	unlimited
N	documents	10^{11} (on disk)
L	avg. # word tokens per document	10^3
M	terms (= word types)	10^8
	avg. # bytes per word token (incl. spaces/punct.)	6
	avg. # bytes per word token (without spaces/punct.)	4.5
	avg. # bytes per term (= word type)	7.5

Hint: You have to make several simplifying assumptions – that's ok, just state them clearly.

提纲

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)

- 一些数字：
  - 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，300万个处理器/核
  - Google每15分钟装入 100,000个服务器.
  - 每年的支出大概是2-2.5亿美元
  - 这可能是世界上计算能力的10%!

- 单机故障 vs. 多机故障
  - 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%，那么整个系统的运行出错概率是多少？
  - 答案: $1-(0.999)^{1000}=63\%$
  - 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

- 维持一台主节点(Master)来指挥索引构建任务-这台主节点被认为是安全的
- 将索引划分成多组并行任务
- 主节点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

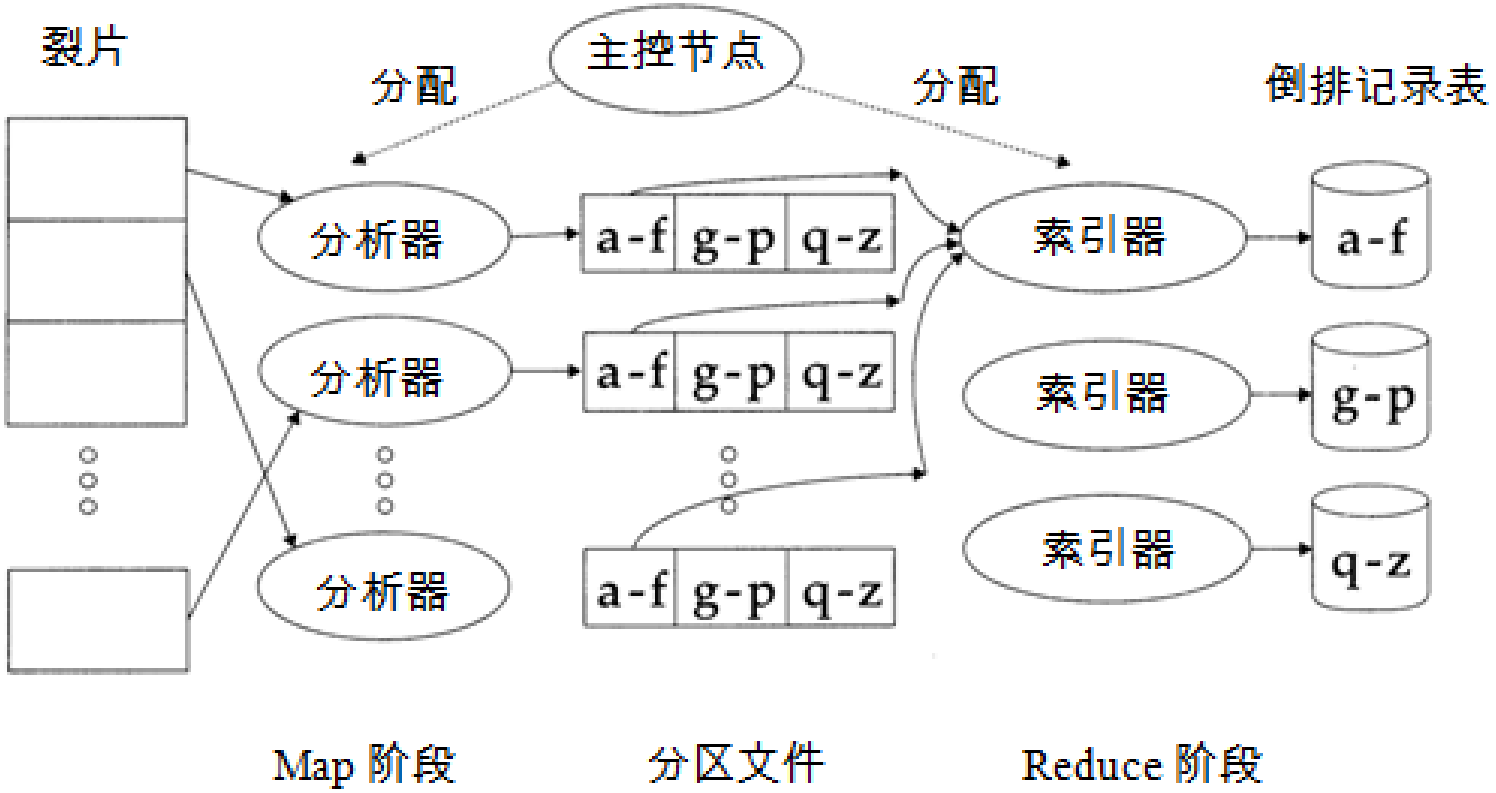
分析器(Parser)

- 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
主控节点
分配
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(2004)是一个鲁棒的简单分布式计算框架，其思想最早由Google提出，一个著名的开源实现是Hadoop
  - 用户不需要关心分布式计算相关的实现细节
- 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

基于MapReduce的索引构建

- Map、Reduce函数的功能：
  - map: input → list(k, v)
  - reduce: (k,list(v)) → output
- 如何应用在索引创建过程中：
  - map: collection → list(termID, docID)
  - reduce: (<termID1, list(docID)>, <termID2, list(docID)>, …) → (postings list1, postings list2, …)

基于MapReduce的索引构建

- 输入：
  - d1 : C came, C c'ed.
  - d2 : C died.
- Map:
  - <C,d1>, <came,d1>, <C,d1>, <c'ed, d1>, <C, d2>, <died,d2>
- Reduce:
  - (<C,(d1,d2,d1)>, <died,(d2)>, <came,(d1)>, <c'ed,(d1)>) →
  - (<C,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <c'ed,(d1:1)>)

Map阶段输出有多少个k-v对？

Reduce阶段输出的倒排记录数有多少个？

课堂练习

- 主节点机传给分析器的任务描述包含什么信息？
- 任务完成后，分析器给主节点机的回传报告中会包含哪些信息？
- 主节点机传给倒排器的任务描述包含什么信息？
- 任务完成后，倒排器给主节点机的回传报告中会包含哪些信息？

基于MapReduce的索引构建

- 上面的索引构建只是一个步骤，实现了按词项分割的索引
- 另一个步骤：将按词项分割的索引转换成按文档分割的索引
  - 按词项分割: 每个节点处理一部分词项索引
  - 按文档分割: 每个节点处理一部分文档集合
- 哪种分割方式更好？
  - 大多数搜索引擎使用文档分割索引
  - 具有更好的负载均衡

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

- 主索引(Main index)+辅助索引(Auxiliary index)
  - 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(对应一个词项)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_1 已经存在)，依此类推……

对数合并算法

```
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())
```

课堂练习

- 给定n=2，$1 \le T \le 30$， 对对数合并算法进行逐步模拟。
- 画出第T=2*k个词条被处理时，所参与合并的索引$I_0,\ldots,I_3$

k	T	I_3	I_2	I_1	I_0
1	2	0	0	0	0
2	4	0	0	0	1
3	6	0	0	1	0

T	I_3	I_2	I_1	I_0
2	0	0	0	0
4	0	0	0	1
6	0	0	1	0
8	0	0	1	1
10	0	1	0	0
12	0	1	0	1
14	0	1	1	0
16	0	1	1	1
18	1	0	0	0
20	1	0	0	1
22	1	0	1	0
24	1	0	1	1
26	1	1	0	0

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
  - $a + 2a + 3a + 4a + \ldots + na = a\frac{n(n+1)}{2} = O(n^2)$
- 因此，对数合并的复杂度比辅助索引方式要低一个数量级

动态索引构建

- 方法1：辅助索引
  - 将新增文档的索引保存在内存中
  - 当内存中的辅助索引大小超过阈值，与磁盘上的主索引合并
    - 增加了I/O操作开销
  - 对数合并
    - 在磁盘上存储多个辅助索引
- 方法2：周期性重建索引
  - 大型搜索引擎通常的做法

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

参考资料

- 《信息检索导论》第4章
- http://ifnlp.org/ir
  - Dean and Ghemawat (2004) 有关MapReduce的原作
  - Heinz and Zobel (2003) 有关SPIMI的原作
  - YouTube视频: Google数据中心

课后练习

- 有待补充