

# 现代信息检索

## Modern Information Retrieval

### 第2讲 索引构建

### Index construction

# 本讲内容

- 怎样构建索引
- 语料通常很大，而服务器内存通常相对较小
  - 在内存有限的情况下的索引构建策略

# 上一讲介绍过的索引构建过程

解析文档得到词条与文档id对 <term, Doc #>

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

# <Term, Doc#>对排序：索引构建关键步骤

在所有文档解析完毕后，倒排索引按照词项排序



Term	Doc #	Term	Doc #
I	1	ambitious	2
did	1	be	2
enact	1	brutus	1
julius	1	brutus	2
caesar	1	capitol	1
I	1	caesar	1
was	1	caesar	2
killed	1	caesar	2
i'	1	did	1
the	1	enact	1
capitol	1	hath	1
brutus	1	I	1
killed	1	I	1
me	1	i'	1
so	2	it	2
let	2	julius	1
it	2	killed	1
be	2	killed	1
with	2	let	2
caesar	2	me	1
the	2	noble	2
noble	2	so	2
brutus	2	the	1
hath	2	the	2
told	2	told	2
you	2	you	2
caesar	2	was	1
was	2	was	2
ambitious	2	with	2

# 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
- 路透社 1995到1996年一年的英语新闻报道
- 本讲使用Reuters RCV1文档集来介绍可扩展的索引构建技术
  - 虽然RCV1语料不算很大，但可以公开获取。我们将以此语料为例说明索引构建所涉及到的问题

# 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](#) | [Reprin](#)

[\[-\] 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

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？

# 基于排序的索引构建方法

- 在构建索引时，每次解析一篇文档
  - 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。
- 如果每个 (termID, docID) 对占用8个字节, 那么处理大规模语料需要大量的空间
- 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
- 以RCV1为例， $T = 100,000,000$ ，这些倒排记录表倒是可以放在一台典型配置的台式计算机的内存中
  - 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



# 可扩展的索引构建

- 内存中的索引构建不具备可扩展性
  - 无法将整个语料塞入内存，然后排序，最后将倒排索引写入磁盘
- 怎样为大规模语料构建索引？
- 需要考虑硬件条件限制
  - 内存，硬盘，速度，等等
- 下面回顾一些关于硬件的基本知识

# 硬件基础知识

IR系统的服务器的典型配置是几G ~ 几十G的内存（小型服务器），也有上百G或1-2T内存的大型服务器。

磁盘空间通常有几T（小型服务器）或10T以上（磁盘阵列）。

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倍以上的差距)

硬盘寻道时间是闲置时间：磁头在定位时不发生数据传输（假设使用的是机械硬盘，这也是目前大规模存储的典型配置）

因此：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

# 一些统计数据(ca. 2008)

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

即便现在，内存中数据传输依然远快于硬盘I/O

# 能否将硬盘用作“内存”？

能否将硬盘当作内存使用，实现一种基于硬盘IO的大规模语料索引构建算法？

不能：对  $T = 100,000,000$  条记录排序太慢 – 太多的寻道操作.

需要一种外部排序算法

# 提纲

- ① 上一讲回顾
- ② 简介
- ③ BSBI算法
- ④ SPIMI算法
- ⑤ 动态索引构建

# BSBI: Blocked sort-based Indexing

## (一种减少寻道操作的排序)

由解析文档得到8字节（倒排）记录：  $(termID, docID)$

需要对 100M 个这样的8字节记录按照termID排序

将所有记录划分为每个大小约为10M的块

能够将10M块全部放入内存

一共有10块

算法基本思想:

收集每一块的倒排记录，排序，将倒排索引写入硬盘

最后将不同的分块合并为一个大的倒排索引

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

BSBIINDEXCONSTRUCTION()

```
1   $n \leftarrow 0$ 
2  while (all documents have not been processed)
3  do  $n \leftarrow n + 1$ 
4       $block \leftarrow \text{PARSENEXTBLOCK}()$ 
5      BSBI-INVERT( $block$ )
6      WRITEBLOCKTODISK( $block, f_n$ )
7  MERGEBLOCKS( $f_1, \dots, f_n; f_{\text{merged}}$ )
```

- 该算法中有一个关键决策就是确定块的大小，一般取决于可用内存



# 对 10个 包含 10M 记录的块排序

首先，将每块读入内存，然后排序：

Quicksort（快速排序）大约需要  $O(N \ln N)$  步

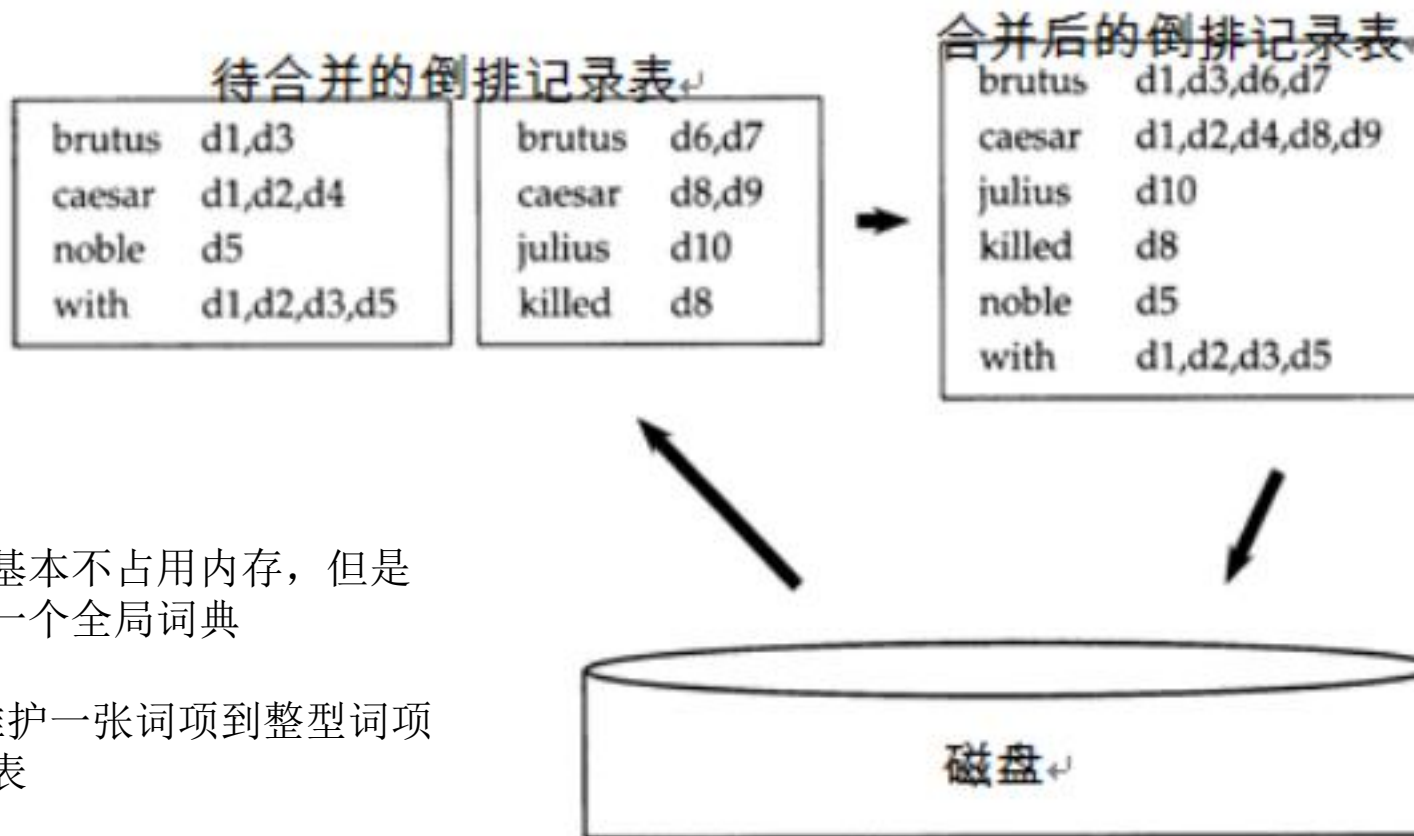
其中，以RCV1为例， $N=10M$

重复上述步骤10次 - 得到10个已排序的 runs，每个包含10M条记录

算法很直接，需要在磁盘中同时保存数据的两份拷贝（合并前与正在合并）

但是可以优化

# 两个块的合并过程



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

# 两个块的合并过程

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



合并后的倒排记录表

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

全局词典

词典： 维护一张词项到整型词项ID的映射表

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

待合并的倒排记录表： 只包含整型ID，没有字符串

# 多个块的合并

- 多项合并(multi-way merge)比两两合并效率更高：从所有块同时读取
  - 同时从所有块读取，并且为每块保持一个读缓冲区(read buffer)，为输出文件（即合并后的索引）保持一个写缓冲区(write buffer)
  - 维护一个待处理termid的优先级队列(priority queue)，每次迭代从队列中选取一个最小的未处理termid
  - 合并不同块中所有的该termID的倒排记录，并写入磁盘

每次迭代均处理较小规模的数据（一个词项的倒排记录）

# 提纲

- ① 上一讲回顾
- ② 简介
- ③ BSBI算法
- ④ SPIMI算法
- ⑤ 动态索引构建

# 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(即待合并倒排记录表)将会变得很大（字符串比整型数空间消耗更大）

# 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（但是对词典排序。实际上由于指针的存在，倒排记录表没有排序的必要），按照它们出现的先后顺序排列
- 在扫描文档的同时，直接在内存中维护一个不断更新的倒排索引
- 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

# 两个块的合并过程

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	3	d10
4	d1, d2, d3, d5	4	d8

Term	Term id	Term	Term id
brutus	1	brutus	1
caesar	2	caesar	2
noble	3	julius	3
with	4	killed	4

待合并的局部词典

合并后的倒排记录表

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

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

合并后的全局词典



# 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.

*output\_file*: 倒排索引; *dictionary*: 词典  
这里假定词典采用哈希表。

在内存中维护局部倒排索引，直至没有更多的空闲内存，将局部索引写入磁盘，释放内存。合并过程与BSBI类似，但此时没有全局词典提供词项-整数ID的映射，合并过程需要进行词项字符串比较

# 两种算法的主要区别

## BSBI算法

在分块索引阶段，BSBI算法维护一个全局Term (String) – Termid (int) 的映射表，局部索引为Termid及其倒排记录表，仍然按词典顺序排序。

## SPIMI算法

分块索引阶段与BSBI算法不同在于建立局部词典和索引，无需全局词典。在合并阶段，将局部索引两两合并，最后产生全局词典建立Term – Termid的映射。

# 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 动态索引构建

# 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 ( $l_0, l_1, \dots$ ) 置于磁盘
- 如果  $z_0$  变得太大 ( $> n$ ), 则将它作为  $l_0$  写到磁盘中(如果  $l_0$  不存在)
- 或者和  $l_0$  合并(如果  $l_0$  已经存在), 并将合并结果作为  $l_1$  写到磁盘中(如果  $l_1$  不存在), 或者和  $l_1$  合并(如果  $l_0$  已经存在), 依此类推.....



LMERGEADDTOKEN(*indexes*,  $Z_0$ , *token*)

```

1   $Z_0 \leftarrow \text{MERGE}(Z_0, \{\text{token}\})$ 
2  if  $|Z_0| = n$ 
3    then for  $i \leftarrow 0$  to  $\infty$ 
4      do if  $l_i \in \text{indexes}$ 
5        then  $Z_{i+1} \leftarrow \text{MERGE}(l_i, Z_i)$ 
6          ( $Z_{i+1}$  is a temporary index on disk.)
7           $\text{indexes} \leftarrow \text{indexes} - \{l_i\}$ 
8        else  $l_i \leftarrow Z_i$  ( $Z_i$  becomes the permanent index  $l_i$ .)
9           $\text{indexes} \leftarrow \text{indexes} \cup \{l_i\}$ 
10         BREAK
11      $Z_0 \leftarrow \emptyset$ 

```

LOGARITHMICMERGE()

```

1   $Z_0 \leftarrow \emptyset$  ( $Z_0$  is the in-memory index.)
2   $\text{indexes} \leftarrow \emptyset$ 
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  $a + 2a + 3a + 4a + \dots + na = a \frac{n(n+1)}{2} = O(n^2)$
  -
- 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)
- **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