

# 信息检索与数据挖掘

---

## 第4章 索引构建与索引压缩

### ——第一讲 索引构建

# 课程内容

- 第1章 绪论
- 第2章 布尔检索及倒排索引
- 第3章 词项词典和倒排记录表
- 第4章 索引构建和索引压缩
- 第5章 向量模型及检索系统
- 第6章 检索的评价
- 第7章 相关反馈和查询扩展
- 第8章 概率模型
- 第9章 基于语言建模的检索模型
- 第10章 文本分类
- 第11章 文本聚类
- 第12章 Web搜索
- 第13章 多媒体信息检索
- 第14章 其他应用简介

# • 第一讲: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)

# 索引构建 (Index construction)

- **思考如下问题：**
  - 我们怎样建立一个索引？
  - 对于给定的计算机内存，我们可以采用怎样的索引构建策略？
- **How do we construct an index?**
- **What strategies can we use with limited main memory?**

# 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

# 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

存储（硬磁盘/SSD、内存）

计算（CPU架构、主频）

I/O（磁盘 $\leftrightarrow$ 内存 $\leftrightarrow$ CPU）

# 硬件基础：存储能力

## • 2007年

- IR系统的服务器通常数GB甚至数十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。

## • 现在



¥339.00

西部数据(WD)蓝盘 1TB SATA6Gb/s 7200转64M 台式机硬盘 (WD10EZEX)

已有253029人评价

☐ 对比 ☐ 关注 ☐ 加入购物车



¥339.00

希捷 (Seagate) 1TB 7200转64M SATA3 台式机硬盘 (ST1000DM003)

已有227111人评价

☐ 对比 ☐ 关注 ☐ 加入购物车



¥459.00

希捷 (Seagate) 2TB 7200转64M SATA3 台式机硬盘 (ST2000DM001)

已有61307人评价

☐ 对比 ☐ 关注 ☐ 加入购物车



¥599.00

希捷 (Seagate) 3TB 7200转64M SATA3 台式机硬盘 (ST3000DM001)

已有22186人评价

☐ 对比 ☐ 关注 ☐ 加入购物车



¥459.00

西部数据(WD)蓝盘 2TB SATA6Gb/s 7200转64M 台式机硬盘 (WD20EZRX)

已有3664人评价

☐ 对比 ☐ 关注 ☐ 加入购物车

# 硬件基础：存储能力(2016)

存储空间



空间范围      价格(每月)

0 - 50TB(含)      ¥0.17 / GB

50TB - 500TB(含)      ¥0.165 / GB

500TB - 5000TB(含)      ¥0.16 / GB

5000TB以上      ¥0.155 / GB

注：单位-元



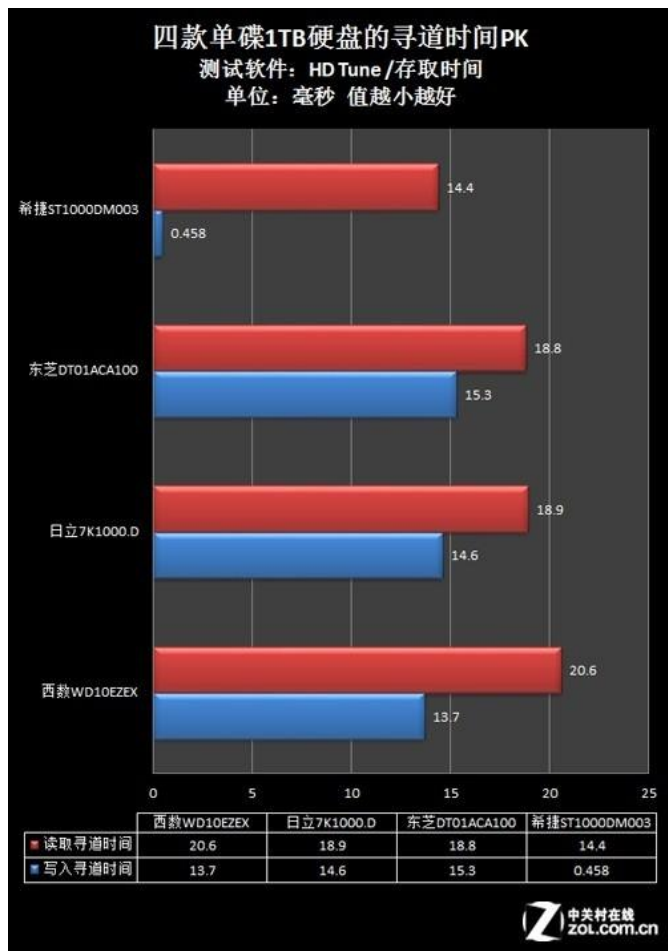
存储包	北京/深圳/上海			杭州		
	1个月	6个月 (买5送1)	12个月 (买9送3)	1个月	6个月 (买5送1)	12个月 (买9送3)
1TB	138	690	1,242	144	720	1,296
10TB	1,375	6,875	12,375	1,436	7,180	12,924
50TB	6,876	34,380	61,884	7,181	35,905	64,629
300TB	37,601	188,005	338,409	39,813	199,065	358,317
500TB	62,669	313,345	564,021	66,355	331,775	597,195



# 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得多。
- 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**

# 硬件基础： I/O能力 HDD参数(2012)



平均寻道时间: 5ms (2007)

每个字节的传输时间: 0.02 $\mu$ s (2007)

# 硬件基础： I/O能力 SSD参数 (2014)

Seq: 连续做读、写硬盘检测(1024K位元组)

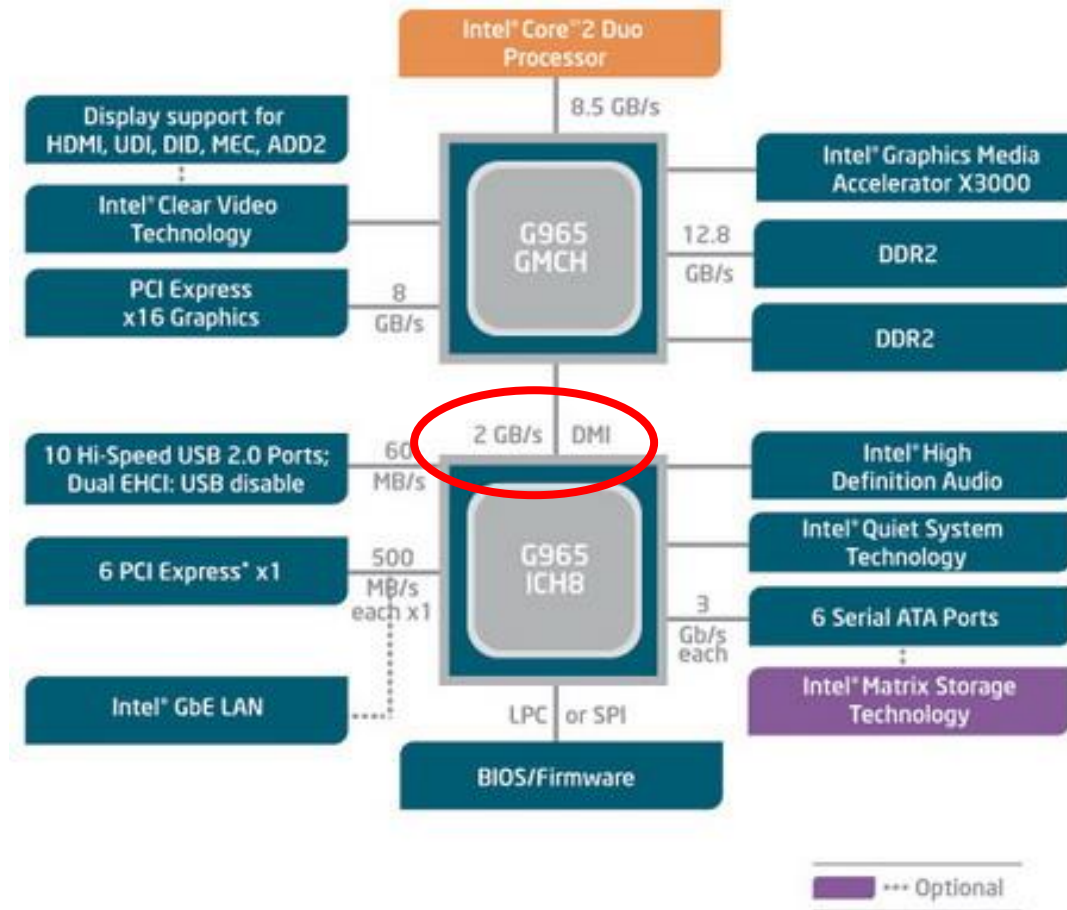
512K: 随机做读、写硬盘检测(512K位元组)

4K: 随机做读、写硬盘检测(4K位元组)

4K QD32: 针对NCQ、AHCI模式做随机读写测



# 硬件基础：计算机I/O能力可能还有很大空间

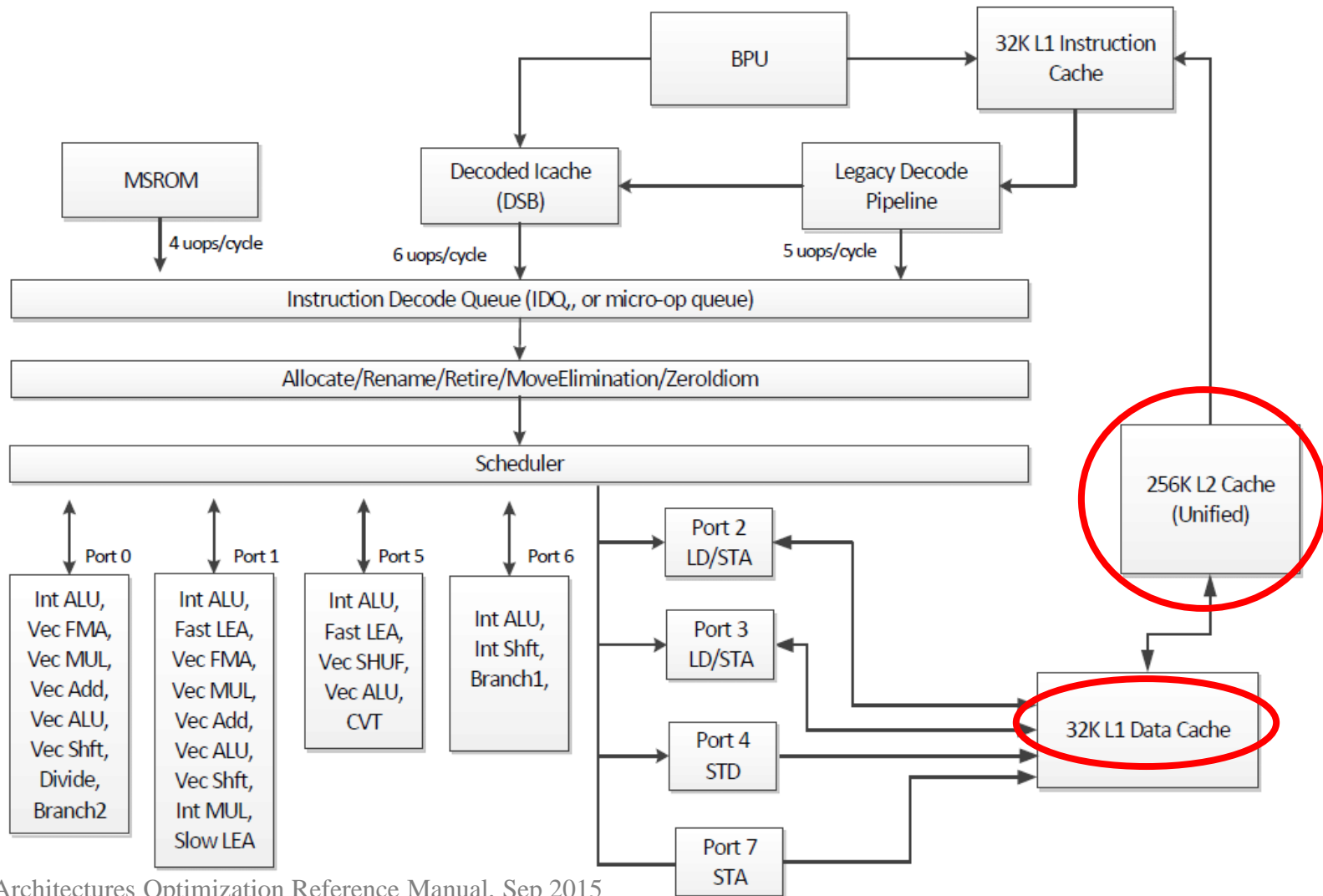


若硬盘**Cache 16MB**?

若16个硬盘的光通道阵列，顺序排列数据块?

# 硬件基础：计算机的I/O能力

## Intel Skylake Microarchitecture



# 硬件基础：计算机的计算能力(2015.10)

- **MIPS: Millions of instructions per second**

- 1978年Intel 8086, 0.33 MIPS@5MHz
- 1989年Intel i486DX, 8.7 MIPS@25MHz
- 2015年Intel i7 6700k, 24182 MIPS@4GHz



- **FLOPS: FLOating-point operations per second**

- 1989年Intel i486DX, 50000 FLOPS
- 2015年Intel i7 6700k(HD Graphics 530, 441.6G FLOPS)
- iPhone6 (PowerVR GX6650, 115.2GFLOPS@300MHz)

手机	价格¥	GPU	FLOPS
小米4	1500	Adreno 330	129.6G
魅蓝2	800	Mali T720	81.6G
P8	3500	Mali-T628	76.8G



# 小结：硬件基础

- 任何时代硬件的能力有上限
  - 存储（硬磁盘/SSD、内存）
  - 计算（CPU架构、主频）
  - I/O（磁盘 $\leftrightarrow$ 内存 $\leftrightarrow$  CPU）
- **IR**算法对存储、计算、I/O的需求？
- 在存储、计算、I/O受限条件下的算法

# 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》作为样例文档集还远远不够。
- **Reuters-RCV1**文档集也并不是真正的足够大, 但它是公开的并且是一个更为合理的样例。我们将使用路透社的**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)。



You are here: [Home](#) > [News](#) > [Science](#) > [Article](#)

Go to a Section: [U.S.](#) [International](#) [Business](#) [Markets](#) [Politics](#) [Entertainment](#) [Technology](#) [Sports](#) [Oddly Enough](#)

## Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

[Email This Article](#) [Print This Article](#) [Reprints](#)



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 meteorological base at Mawson Station on July 25.

# Reuters-RCV1语料：统计数据

符号	含义	值
N	文档总数	800,000
L	每篇文档的平均词条数目	200
M	词项(Term)总数	400,000
	每个词条(Token)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) 总数目	100,000,000

每个词条占4.5字节 VS. 每个词项占7.5字节：为什么？

# Reuters-RCV1语料：索引构建中的临时文件

- $N=800,000$        $2^{20} > N > 2^{16}$        $\rightarrow$  文档ID需32bit
- $T=100,000,000$        $2^{20+7} > N > 2^{16}$        $\rightarrow$  词条ID需32bit
- 存储“词条ID-文档ID”需要
- $T \times (32\text{bits} + 32\text{bits}) = \mathbf{0.8\text{G}\text{Bytes}}$

符号	含义	值
N	文档总数	800,000
T	词条 (Token) 总数目	100,000,000

我们需要对0.8GB的ID对进行排序！  
而实际语料库要比RCV1大

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	docID	term	docID	term	doc. freq.	→	postings lists
I	1	ambitious	2	ambitious	1	→	2
did	1	be	2	be	1	→	2
enact	1	brutus	1	brutus	2	→	1 → 2
julius	1	brutus	2	capitol	1	→	1
caesar	1	capitol	1	caesar	2	→	1 → 2
I	1	caesar	1	caesar	2	→	1 → 2
was	1	caesar	2	did	1	→	1
killed	1	caesar	2	enact	1	→	1
i'	1	did	1	hath	1	→	2
the	1	enact	1	I	1	→	1
capitol	1	hath	1	i'	1	→	1
brutus	1	I	1	it	1	→	2
killed	1	I	1	julius	1	→	1
me	1	i'	1	killed	1	→	1
so	2	it	2	let	1	→	2
let	2	julius	1	me	1	→	1
it	2	killed	1	noble	1	→	2
be	2	killed	1	so	1	→	2
with	2	let	2	the	2	→	1 → 2
caesar	2	me	1	told	1	→	2
the	2	noble	2	you	1	→	2
noble	2	so	2	was	2	→	1 → 2
brutus	2	the	1	with	1	→	2
hath	2	the	2				
told	2	told	2				
you	2	you	2				
caesar	2	was	1				
was	2	was	2				
ambitious	2	with	2				

# 词典大小、倒排记录大小

- $M=400,000$        $2^{19} > N > 2^{16}$        $\rightarrow$  词项ID需32bit
  - 词典大小:  $M * 32\text{bits} = 1,600,000\text{Bytes} = 1.6\text{MBytes}$
- $N=800,000$        $2^{20} > N > 2^{16}$        $\rightarrow$  文档ID需32bit
- $L * N = 160,000,000$        $2^{20+8} > N > 2^{20+7}$ 
  - 倒排记录: 约  $L * N * 32\text{bits} = 640,000,000\text{Bytes} = 0.64\text{GBytes}$ 
    - 不考虑倒排记录存储数据结构的额外开销

符号	含义	值
N	文档总数	800,000
L	每篇文档的平均词条数目	200
M	词项(Term)总数	400,000

## 回顾：词条化

### 实际排序的不是词条ID而是词条

- 将每篇文档转换成一个个词条的列表并加上文档的ID

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

## 回顾：词条化的关键步骤

比较词条可能需要更多的CPU周期

- 在所有文档被转换后，倒排表按照词项的字母顺序进行排序

我们关注这一排序过程。  
我们有100M 的词条需要排序。

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

# 扩展索引构建

- 在内存中进行索引构建并不能扩展。
- 我们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

- 我们在建立索引过程中，需要依次分析所有的文档。
  - 索引构建过程中，我们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 $T=100,000,000$ 
  - 我们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：我们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

# 在硬盘中采用同样的算法？

- 对于大型的语料库，我们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。
- 我们需要一个**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。
- 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 $T=100M$ 条记录。



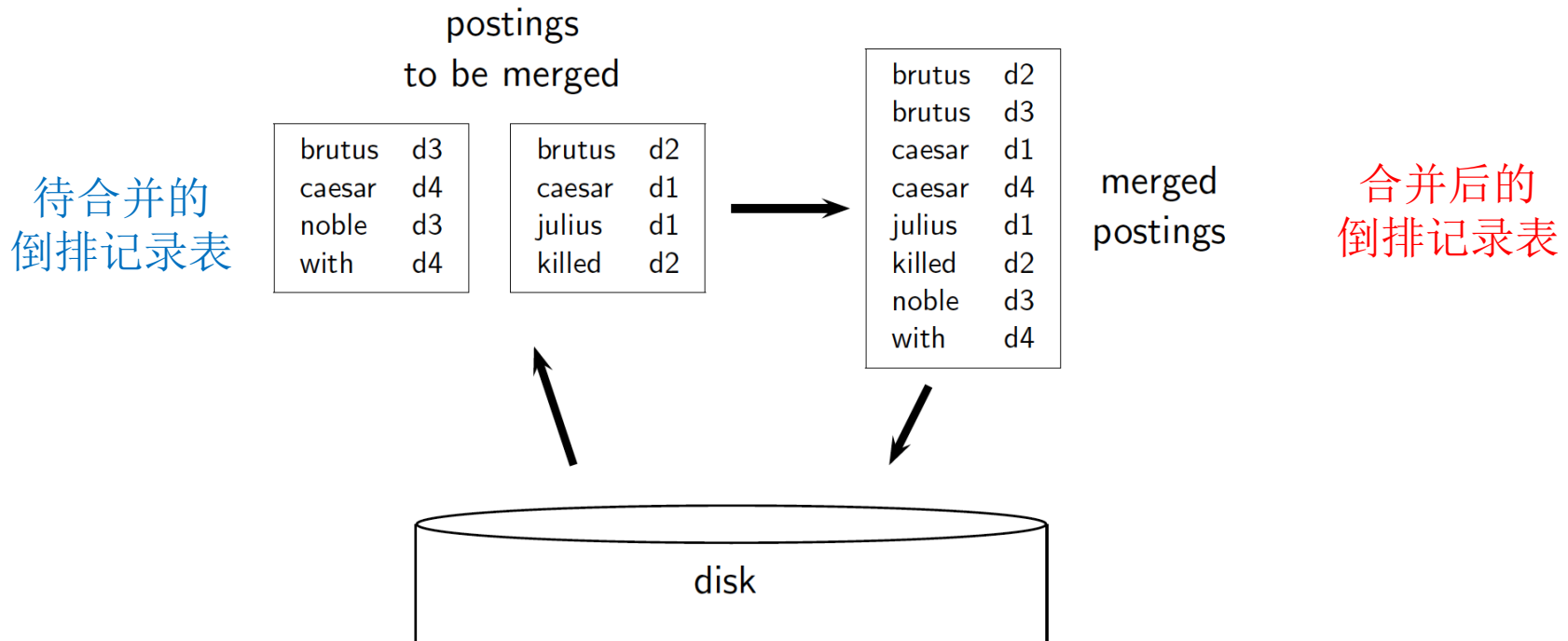
假如每次比较需要2次的磁盘寻道，对 $N$ 条记录进行排序需要 $N \log_2 N$  次比较，我们需要花费多少的时间？

# 基于块的排序索引算法

## BSBI: Blocked sort-based Indexing

- 该算法的基本思想:

- 对每一个块都生成倒排记录, 并排序, 写入硬盘中。
- 然后将这些块合并成一个长的排序好的倒排记录。



# 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)
- 这些数据是在我们分析文档时生成。
- 我们需要对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。
  - 我们开始加载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**
- **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)

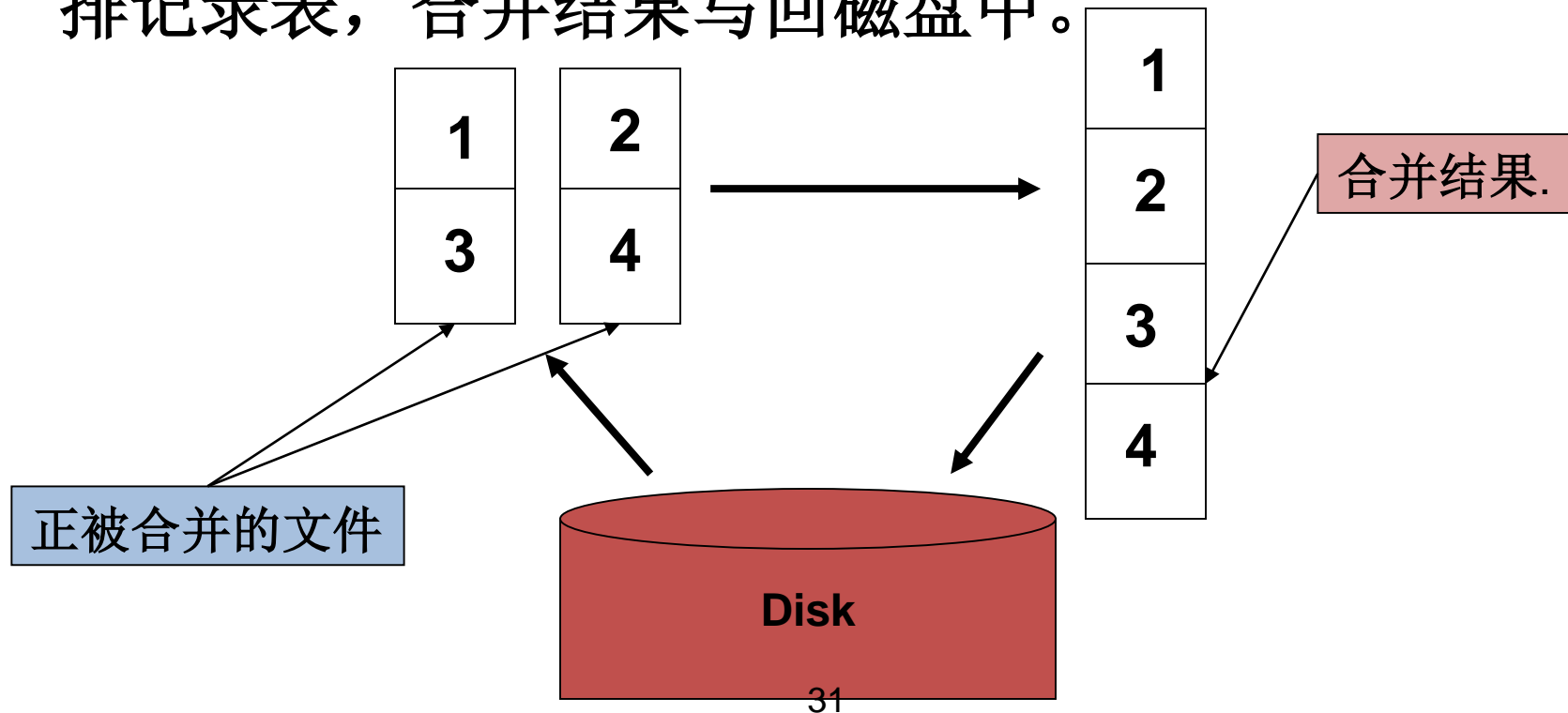
- $f_1, f_2, \dots$  合并为  $f_{\text{merged}}$

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( $f_1, \dots, f_n; f_{\text{merged}}$ )
```

# 如何合并排序结果？

- 可以进行二分合并，可以产生一个 $\log_2 10$  , 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？

- 但是一个 $n$ -路的合并会更加高效，我们可以同时读取所有的数据块。
- 内存中维护了为10个块准备的读缓冲区和一个为最终合并索引准备的写缓冲区，这样我们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

- 我们的假设是：我们能够将词典存入内存中。
- 我们需要该词典(**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，我们可以采用<**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, 我们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: 倒排

```
SPIMI-INVERT(token_stream)
1  output_file = NEWFILE()
2  dictionary = NEWHASH()
3  while (free memory available)
4  do token  $\leftarrow$  next(token_stream)
5      if term(token)  $\notin$  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  $\leftarrow$  SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

- 块合并操作与BSBI算法类似

# 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规模的索引构建(不要在家里尝试! )**
  - 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
- **我们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
- 这将会是全球所有计算能力的**10%**吗？

# 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%**
- 我们可以试着计算一下：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(Cluster)**中的主控节点指挥索引构建工作。
  - 我们认为主控节点是“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

- 我们采用两组不同的并行任务
  - Parsers 分析器
  - Inverters 倒排器(中文版教材写成索引器)
- 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： 基于词项的分割、基于文档的分割

(a) Document partitioning

		Documents								
Terms		D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>	D <sub>6</sub>	D <sub>7</sub>	D <sub>8</sub>	D <sub>9</sub>
	T <sub>1</sub>	X		X	X		X			X
	T <sub>2</sub>		X			X				
	T <sub>3</sub>		X	X					X	
	T <sub>4</sub>				X			X		
	T <sub>5</sub>	X					X			X
	T <sub>6</sub>	X						X	X	
	T <sub>7</sub>		X		X		X			
	T <sub>8</sub>			X					X	
		Node 1			Node 2			Node 3		

(b) Term partitioning

		Documents								
Terms		D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>	D <sub>6</sub>	D <sub>7</sub>	D <sub>8</sub>	D <sub>9</sub>
	T <sub>1</sub>	X		X	X		X			X
	T <sub>2</sub>		X			X				
	T <sub>3</sub>		X	X					X	
	T <sub>4</sub>				X			X		
	T <sub>5</sub>	X					X			X
	T <sub>6</sub>	X						X	X	
	T <sub>7</sub>		X		X		X			
	T <sub>8</sub>			X					X	
		Node 1			Node 2			Node 3		

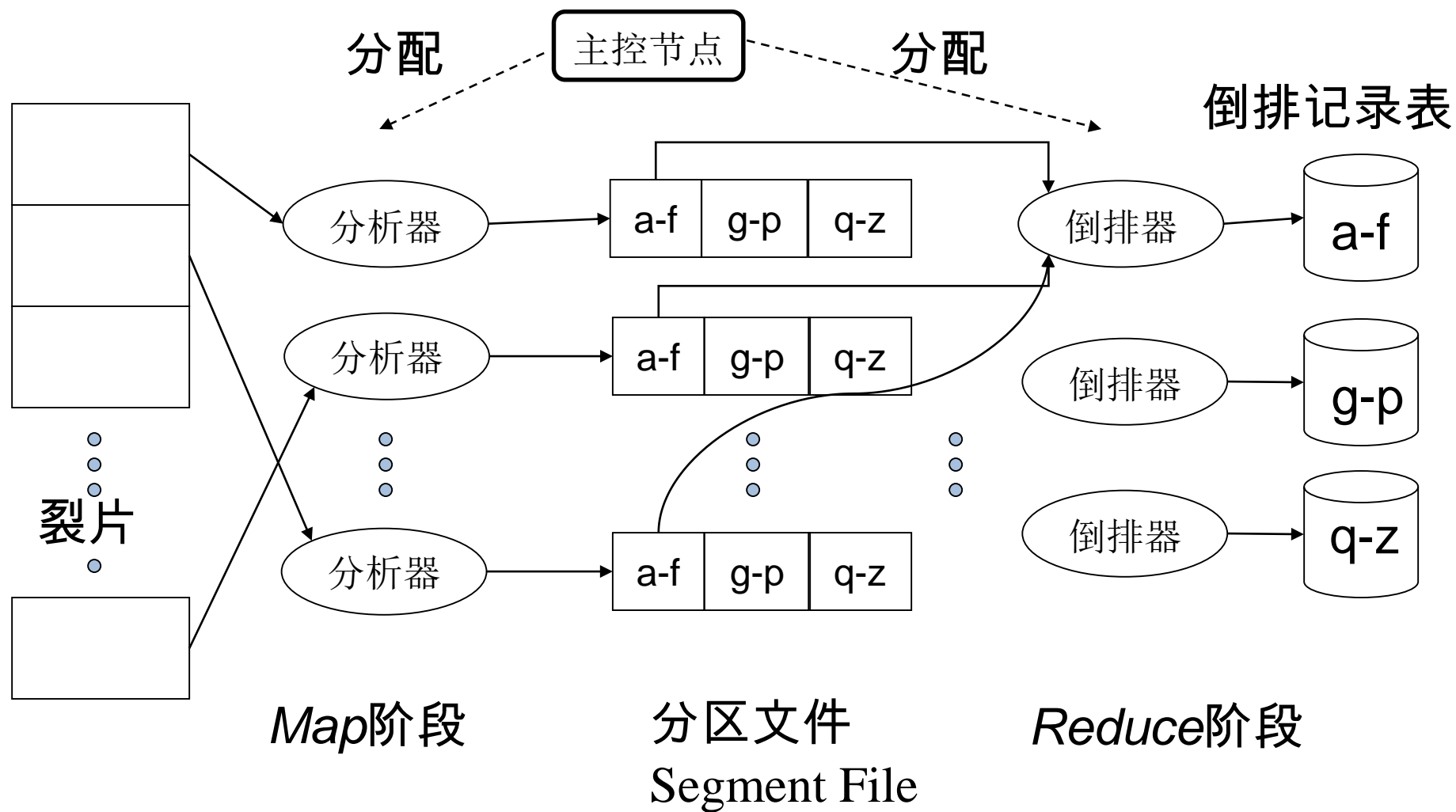
# 分析器 Parsers

- 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>对分成 $j$ 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

- 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



# MapReduce

- 刚刚我们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。
  - 我们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。

# MapReduce

- 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  $\rightarrow$  list(k,v)    Reduce: (k, list(v))  $\rightarrow$  输出

- **索引构建中上述架构的实例化**

Map: Web文档集  $\rightarrow$  list(词项ID, 文档ID)

Reduce: ( $\langle$ 词项ID<sub>1</sub>, list(文档ID) $\rangle$ ,  $\langle$ 词项ID<sub>2</sub>, list(文档ID) $\rangle$ , ...)  
 $\rightarrow$  (倒排记录表1, 倒排记录表2, ...)

- **教材中索引构建的例子**

P53图4-6

Casesar简写成C、conquered简写成c'ed

Map:  $d_2$ : C died.  $d_1$ : C came, C c'ed.

$\rightarrow$  ( $\langle$ C,  $d_2$  $\rangle$ ,  $\langle$ died,  $d_2$  $\rangle$ ,  $\langle$ C,  $d_1$  $\rangle$ ,  $\langle$ came,  $d_1$  $\rangle$ ,  $\langle$ C,  $d_1$  $\rangle$ ,  $\langle$ c'ed,  $d_1$  $\rangle$ )

Reduce: ( $\langle$ C, ( $d_2, d_1, d_1$ ) $\rangle$ ,  $\langle$ died, ( $d_2$ ) $\rangle$ ,  $\langle$ came, ( $d_1$ ) $\rangle$ ,  $\langle$ c'ed, ( $d_1$ ) $\rangle$ )

$\rightarrow$  ( $\langle$ C, ( $d_1:2, d_2:1$ ) $\rangle$ ,  $\langle$ died, ( $d_2:1$ ) $\rangle$ ,  $\langle$ came, ( $d_1:1$ ) $\rangle$ ,  $\langle$ c'ed, ( $d_1:1$ ) $\rangle$ )

《Julius Caesar》/《尤利乌斯·凯撒》，莎士比亚以罗马故事为题材的三出戏之一

I came, I saw, I conquered

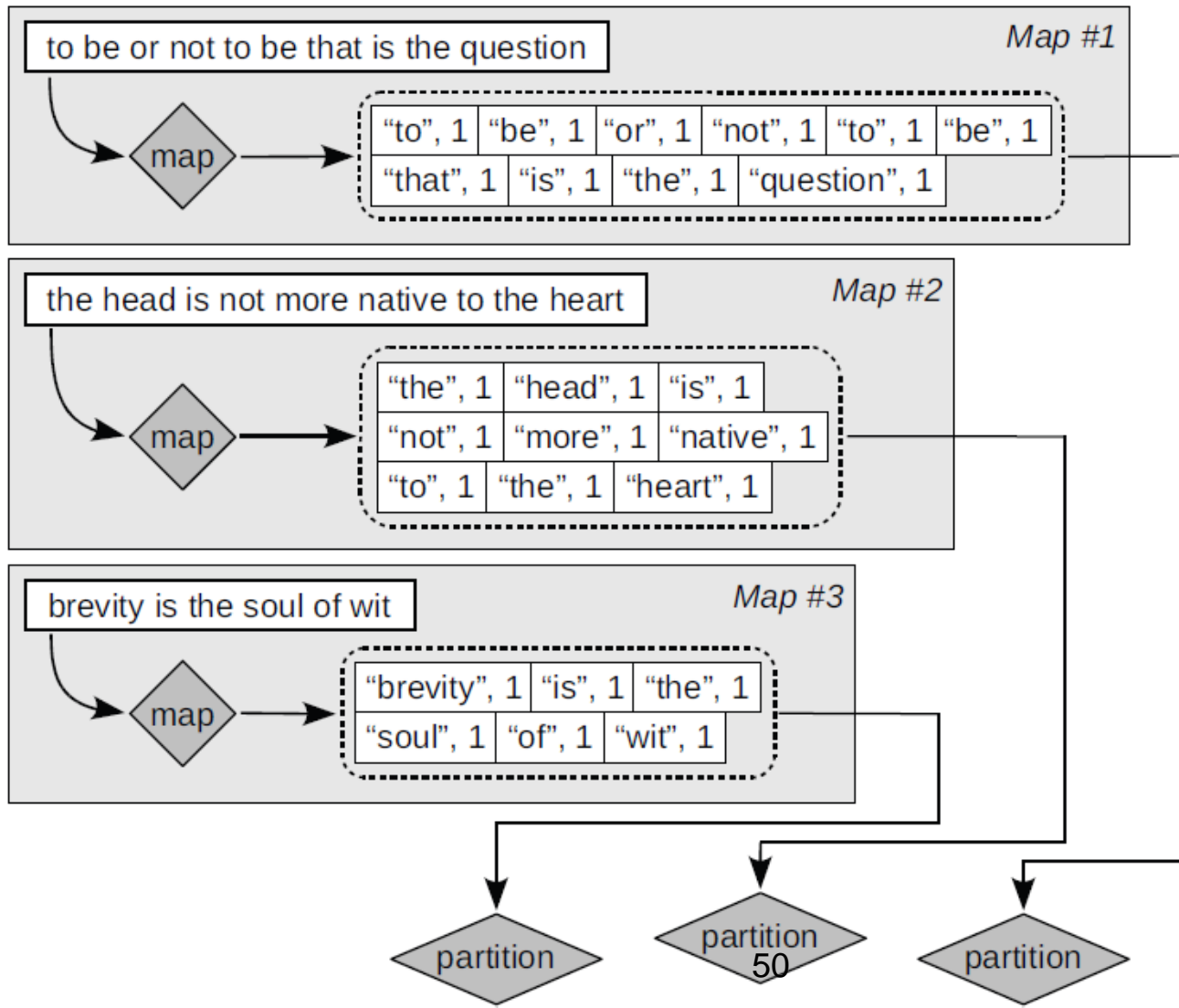
# 一个简单的例子： Map阶段

莎士比亚《哈姆雷特》

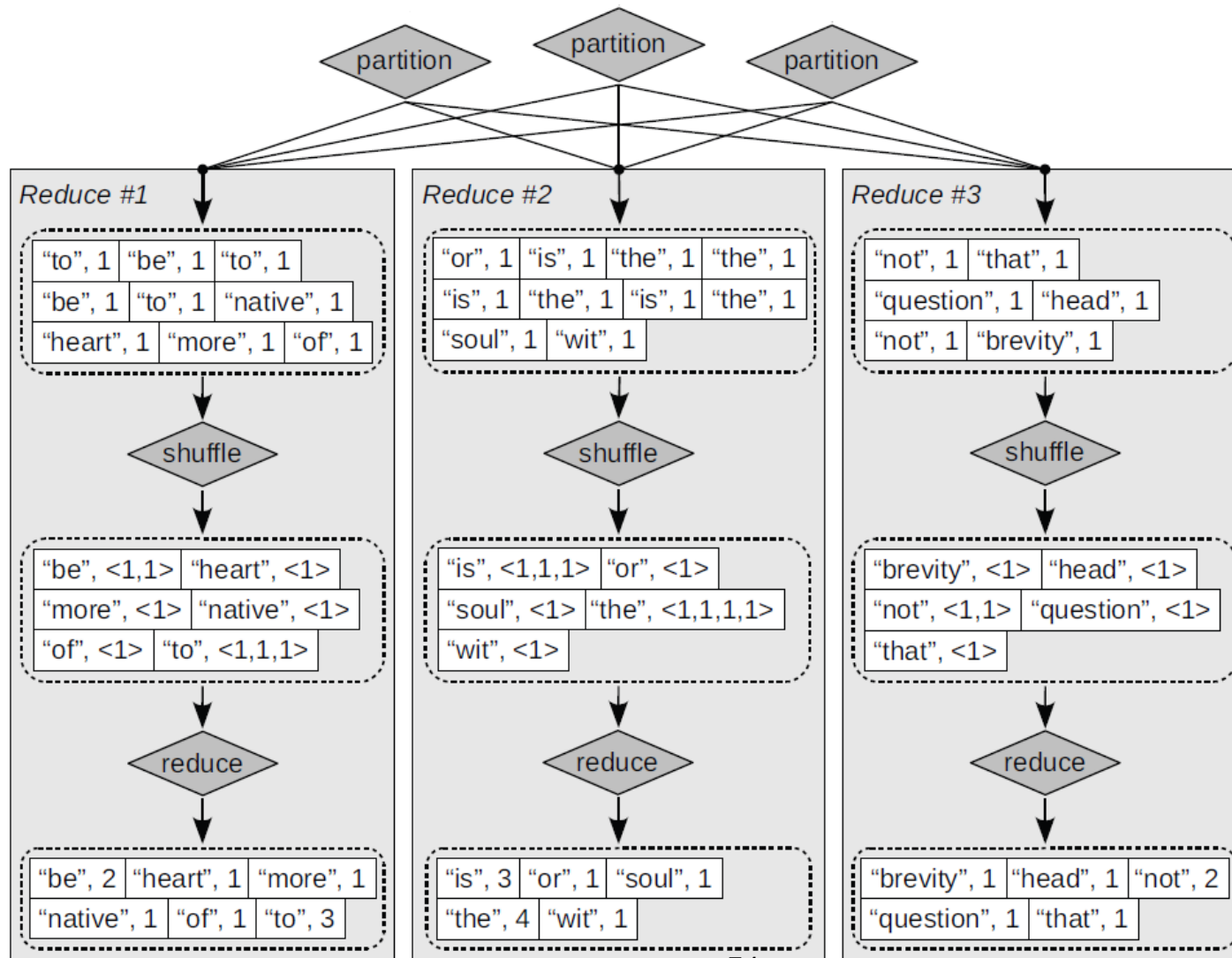
To be, or not to be: that is the question

the head is not more native to the heart("native" = loyal)

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：
  - 文档会不断地加入进来
  - 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

# 最简单的方法

- 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
- 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

# 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。
  - 但我们需要非常多的倒排文件 — 对文件系统来说是低效的。
- 以后课程中我们都假设：索引是一个大的文件。
- 现实中：我们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，每个都是前一个的两倍大小。
- 将最小的 $Z_0$ 存储在内存中
  - $Z_0$ 中有 $2^0 \cdot n$ 个倒排记录
- 将较大的那些( $I_0, I_1, \dots$ )存储在磁盘中
  - $\log_2(T/n)$ 个索引， $I_0, I_1, I_2, \dots$ ,
  - 每个索引大小分别是 $2^0 \cdot n, 2^1 \cdot n, 2^2 \cdot n, \dots$
- 当 $Z_0$ 达到上限 $n$ 时，将它写入磁盘 $I_0$ 中
- 当 $Z_0$ 下一次达到上限时，它会和 $I_0$ 合并，生成 $Z_1$ 
  - 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 $Z_2$
- 以此类推...



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          $Z_0 \leftarrow \emptyset$ 

```

P55图4-7

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

- 辅助索引和主索引：因为每次合并都会处理倒排记录，所以索引构建时间为 $O(T^2)$
- 对数合并：每个倒排记录被合并了 $O(\log T)$ 次，所以复杂度为 $O(T \log T)$
- 所以对于索引构建来说，对数合并是非常高效的
- 但是查询过程现在需要用到合并的 $O(\log T)$ 个索引
  - 如果我们只有一个主索引和一个辅助索引，则为 $O(1)$

# 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，我们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，我们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，新的专题Web网页
- 但是他们也会周期性地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

# 其他索引类型

- 包含位置信息的索引

- 是同样的排序问题 — 只是带来了更大的数据开销

Why?

- 建立一个字符的n-gram索引:

- 当文档分析完后，列举所有的n-gram词项
  - 对每一个n-gram词项，需要一个指针指向所有包含它的词典词项 — “倒排记录”
  - 注意到，在分析文档过程中，同样的“倒排记录项”会重复生成 — 需要高效的hash算法来跟踪它
    - 例如，在词项**deciduous**中出现的3-gram “**uou**” 也会在其他包含词项**deciduous**的文档中被发现。
    - 对于每个词项，我们只需要处理一次。

# 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**

# 课后练习

- 教材第四章： 习题2, 3, 4, 10
- 教材第二章： 习题4, 6, 10 （上次课相关内容）