

信息组织与检索

第4讲：索引构建

主讲人：张蓉

华东师范大学 数据科学与工程学院

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

课后作业

- 请调查现有商业搜索引擎对通配符查找的支持情况。
- 常见搜索引擎：谷歌、百度、搜狗、美团、亚马逊等
- 要点：
 - 是否支持，如支持是否完全支持。
 - 支持与否的原因分析。
 - 400字之内

上一讲内容

- 词典的数据结构：
 - 哈希表 vs. 树结构 (各有什么优缺点?)
- 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
 - (怎么完成nom* / *nom*) 轮排怎么做? K-gram(K=2)?
 - 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

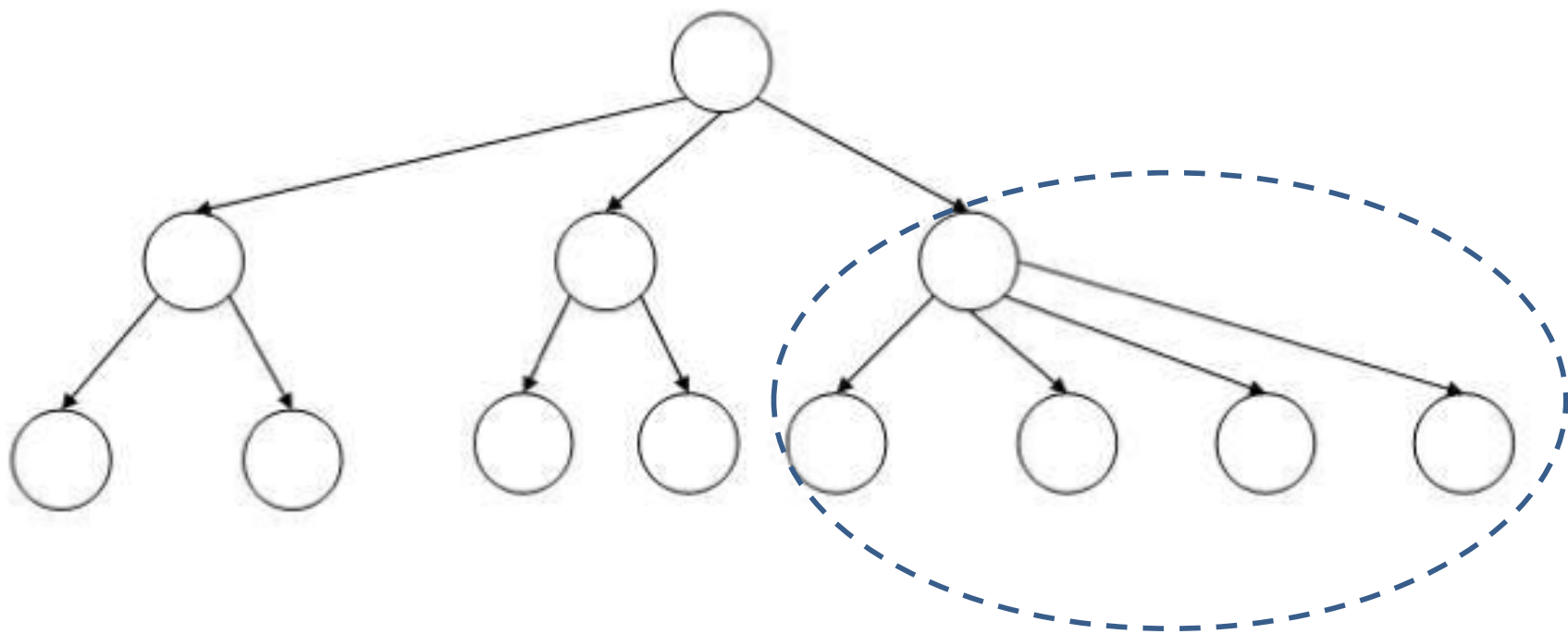
- 哈希表：
 - 定位速度快，常数时间
 - 不宜支持动态变化的词典
 - 不支持前缀查询
- 树结构：二叉树、B-树等等
 - 定位速度为指数时间
 - 二叉(平衡)树支持动态变化，但是重排代价大。
B-树能否缓解上述问题
 - 支持前缀查询

找“刘德华”？
找“刘*”？

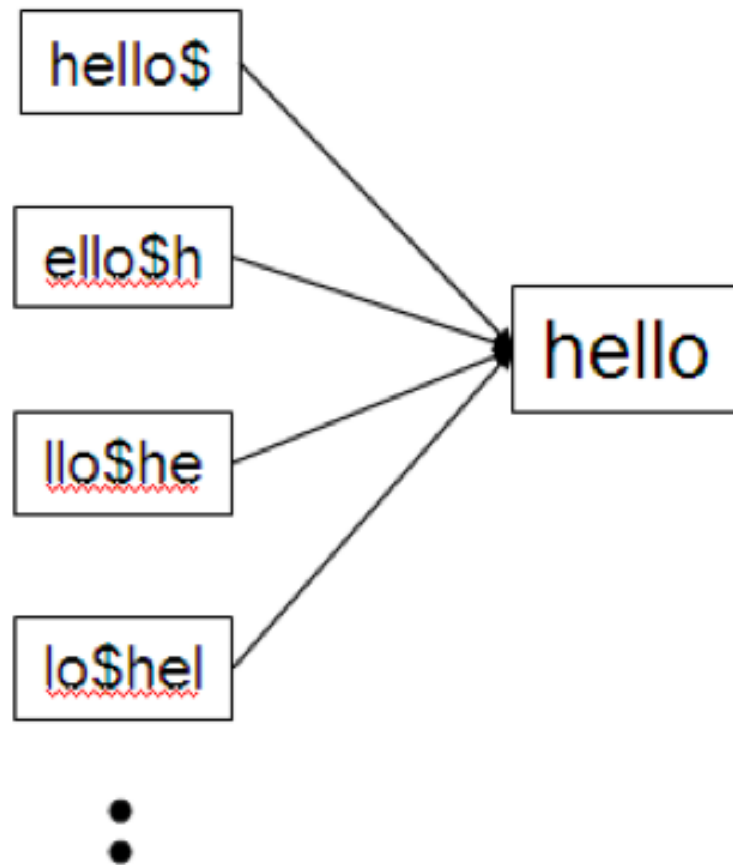
B树/B-树/B+树

- 二叉搜索树
- B树/B-树：一种多路搜索树（并不是二叉的）
- B+树：B+树是B-树的变体
 - B+的搜索与B-树也基本相同，区别是B+树只有达到叶子结点才命中（B-树可以在非叶子结点命中）

基于B-树的词典查找

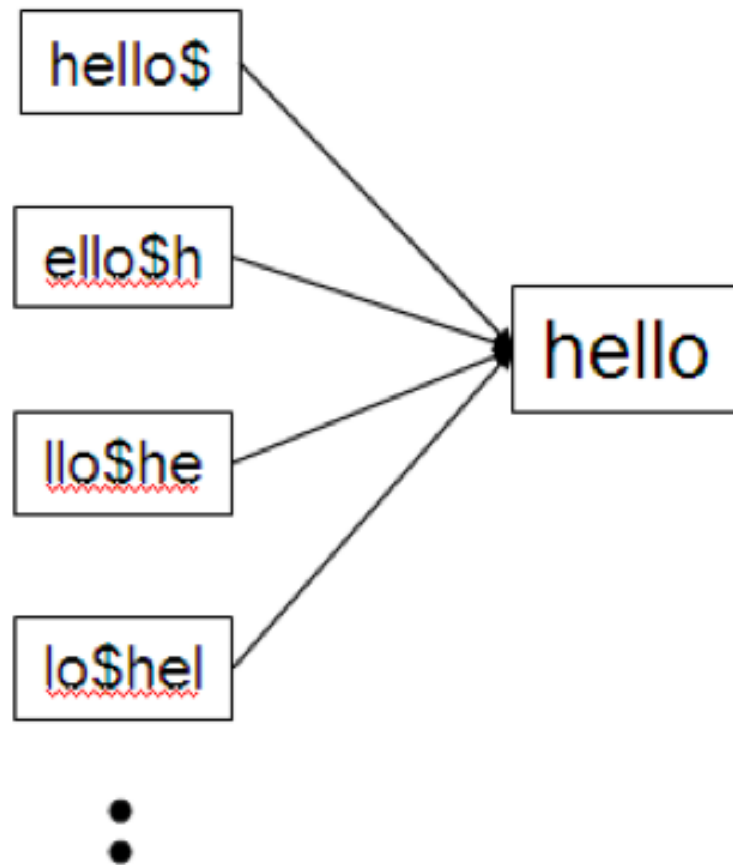


基于轮排索引的通配查询处理



- 查询:
- 对 X, 查找 ?
 - 对 X*, 查找 ?
 - 对 *X, 查找 ?
 - 对 *X*, 查找 ?
 - 对 X*Y, 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*

b*k -> k\$b*

book/back/

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: 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)

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)

校对

- 亡羊修牢->亡羊牢，亡羊买牢，亡羊补牢，。。。

[百度一下](#)

[网页](#) [资讯](#) [视频](#) [图片](#) [知道](#) [文库](#) [贴吧](#) [采购](#) [地图](#) [更多»](#)

百度为您找到相关结果约1,290,000个 [搜索工具](#)

 以下包含 [亡羊补牢](#) 的搜索结果。仍然搜索: [亡羊修牢](#)

[亡羊补牢_百度百科](#)



亡羊补牢，汉语成语，拼音是wáng yáng bǔ láo，意思是羊逃跑了再去修补羊圈，还不算晚。比喻出了问题以后想办法补救，可以防止继续受损失。出自《战国策·楚策》。

[成语故事](#) [成语出处](#) [成语用法](#)

<https://baike.baidu.com/>

[亡羊补牢_百度图片](#)

亡羊补牢: [卡通图片](#) [人物](#) [配图](#) [拼音](#) [插图](#) [看图猜成语](#) [背景图片](#) [更多](#)



- 到目前为止
 - 索引/查询，我们都学习了一遍
 - 可以自己做一个系统了
 - 但是，对于大规模文件集合，会出现啥问题？

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

- 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

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价格更便宜

一些统计数据(ca. 2008)

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	$0.02 \mu\text{s} = 2 \times 10^{-8}$ 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)	$0.01 \mu\text{s} = 10^{-8}$ 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，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



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

<i>N</i>	文档数目	800,000
<i>L</i>	每篇文档的词条数目	200
<i>M</i>	词项数目(= 词类数目)	400,000
	每个词条的平均字节数 (含空格和标点)	6
	每个词条的平均字节数 (不含空格和标点)	4.5
	每个词项的平均字节数	7.5
<i>T</i>	无位置信息索引中的倒排记录数目	100,000,000

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



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		term	docID
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

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
- 好比大家脑袋（内存）记不住很多事情，要记笔记（硬盘）一样

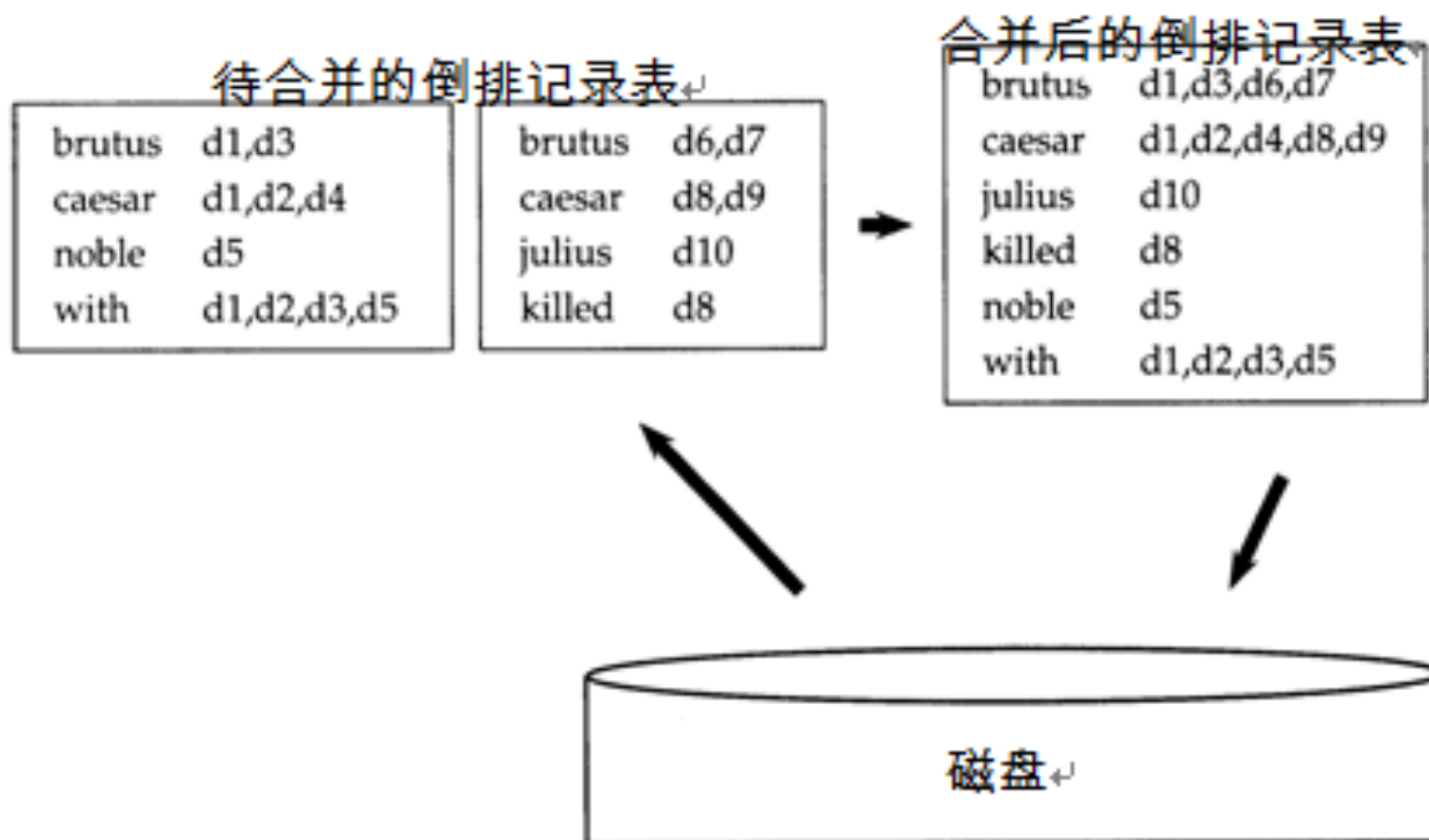
是否在磁盘上采用同样的算法？

- 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（是什么？）

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 ($12 \times 10M = 120M$)
 - 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



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

BSBINDEXCONSTRUCTION()

```
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

BSBI算法

- 将文档中的词进行ID的映射，这里可以用hash的方法去构造
- 将文档分割成大小相等的部分。
- 将每部分按照词ID对文档ID的方式进行排序
- 将每部分排序好后的结果进行合并，最后写出到磁盘中。
- 然后递归的执行，直到文档内容全部完成这一系列操作。
- 必须进行词的ID映射，为什么？
 - 1.排序，2.全局统一ID，3.ID化省空间。

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

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

多智时代

- 对于Web数据级别的数据建立索引：
 - 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

- 全球最大搜索引擎，拥有以太级别的数据，依靠的是遍布全球的15个数据中心：美洲9个、欧洲4个、和亚洲2个。





Google 数据中心

- 全球最大搜索引擎，拥有以太级别的数据，依靠的是遍布全球的15个数据中心：美洲9个、欧洲4个、和亚洲2个。
- 这些数据库的选址标准大约包括：1、大量廉价电力；2、绿色能源，更注重可再生能源；3、靠近河流或湖泊（因设备冷却需要大量水源）；4、用地广阔（隐秘性和安全性）；5、和其他数据中心的距离（保证数据中心间的快速链接）；6、税收优惠。



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: 
- 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: 

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

被索引文本

```
T0="MapReduce is simple"
T1="MapReduce is powerful is simple"
T2="Hello MapReduce bye MapReduce"
```

索引文件

```
"MapReduce": {(T0,1),(T1,1),(T2,2)}
"is":        {(T0,1),(T2,2)}
"simple":     {(T0,1),(T1,1)}
"powerful":  {(T1,1)}
"Hello":     {(T2,1)}
"bye":       {(T2,1)}
```

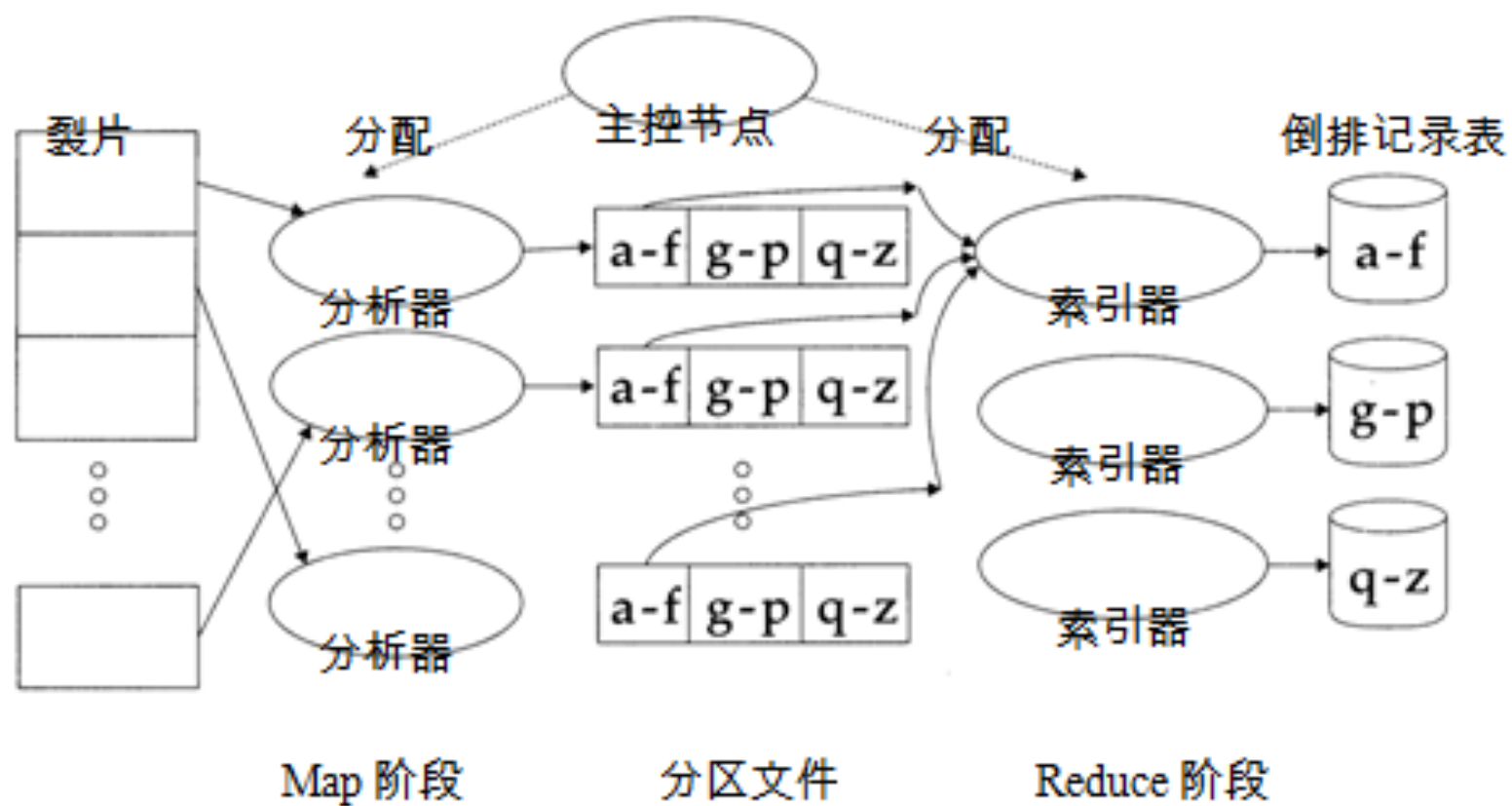
分析器 (Parser)

- 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



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...
- ...不一定需要在分布式处理的部分编写代码
- 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

→ $list(\text{词项ID}, \text{文档ID})$

Reduce: $(\langle \text{文档ID}_1, list(\text{docID}) \rangle, \langle \text{文档ID}_2, list(\text{docID}) \rangle, \dots)$

→ (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.

→ $(\langle C, d_2 \rangle \langle \text{died}, d_2 \rangle, \langle C, d_1 \rangle \langle \text{came}, d_1 \rangle \langle C, d_1 \rangle, \langle \text{c'ed}, d_1 \rangle)$

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

→ $(\langle C, (d_1:2, d_2:1) \rangle, \langle \text{died}, (d_2:1) \rangle, \langle \text{came}, (d_1:1) \rangle, \langle \text{c'ed}, (d_1:1) \rangle)$

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

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。
 - 怎么更新？

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
 - \rightarrow 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, \dots) 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.....

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

- $a + 2a + 3a + 4a + \dots + na = a \frac{n(n+1)}{2} = O(n^2)$

- 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)
- 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