信息检索与Web搜索

第5讲 索引构建

Index construction

授课人：高曙明

*改编自“现代信息检索”网上公开课件（http://ir.ict.ac.cn/~wangbin）

相关硬件基础知识

- 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)
- 硬盘寻道需要时间，即寻道时间，期间不发生数据传输
- 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内存几个GB或几十GB，硬盘数百G或者上T
- 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(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 μs = 2×10^{-8} s
	处理器时钟频率	Hz
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

- 路透社 1995到1996年一年的英语新闻报道文档集，包括政治、贸易、体育、科技等话题

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

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？

回顾：倒排索引组成


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

回顾：倒排索引构建的基本步骤

term	docID
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	docID
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

图 1-4

① 扫描一遍文档集合得到所有的词项——文档二元组

② 以词项为主键，以文档ID为次键进行排序

③ 将每个词项对应的所有文档ID组织成倒排记录表

④ 计算文档频率等统计量

问题：该方法对大规模文档集不适用

基于排序的分块索引构建BSBI

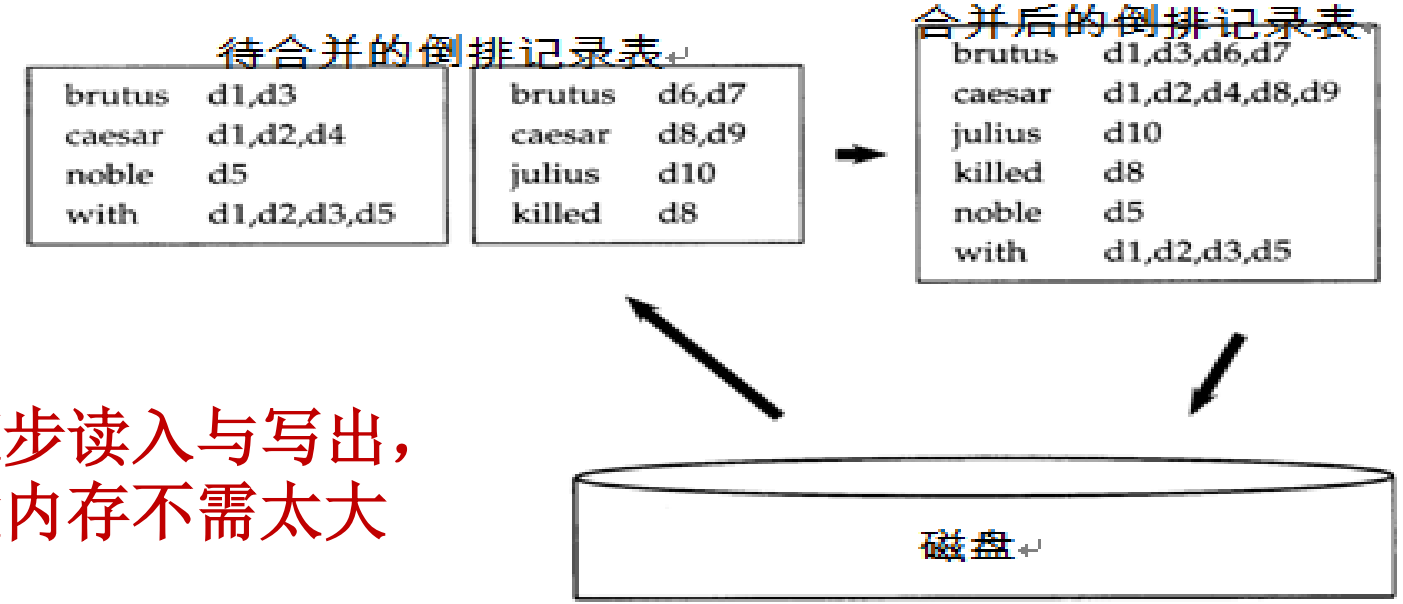
- **基本思想：**对大规模文档集的索引构建进行分而治之
- **算法步骤：**
  1. 将文档集分割成若干大小相当的部分
  2. 将每个部分的词项ID-文档ID二元组排序
  3. 将每个部分的倒排记录表写到磁盘中
  4. 将所有的中间结果合并成整个文档集的倒排索引

BSBI算法

```
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)
```

- 一个关键决策：如何确定块的大小
  - 能够方便加载到内存
  - 能够在内存中进行快速排序

两个块的合并过程



逐步读入与写出，
故内存不需太大

基于BSBI的RCV1索引构建

- 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
- 基本过程:
  - 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

内存式单遍扫描索引构建SPIMI

- **基本思想：**对大规模文档集的索引构建进行分而治之
- **算法特点**
  - 对每个块都产生一个独立的词典，使用词项而不是其ID，不需要统一的词典
  - 增量式动态形成倒排记录表，避免对所有词项—文档ID二元组进行排序
  - 在处理过程中形成块，而不是一开始分块，使分块更合理

SPIMI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.

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

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

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.

分布式索引构建

- **分布式计算环境：**计算机集群，由很多台日用计算机组成，每台计算机随时可能出故障
- **基本思想**
  - 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
  - 将索引构建划分成多组并行任务
  - 主机将把每个任务分配给空闲机器来执行
  - 当发现某台计算机有问题时，将其任务重新分配
- **分布式策略：**基于词项分割；基于文档分割

Google数据中心(2007Gartner)

- 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%，整个系统的正常运行概率是多少？ 63%
- 假定一台服务器3年后会失效，对于100万台服务器，机器失效的平均间隔大概是多少？不到2分钟

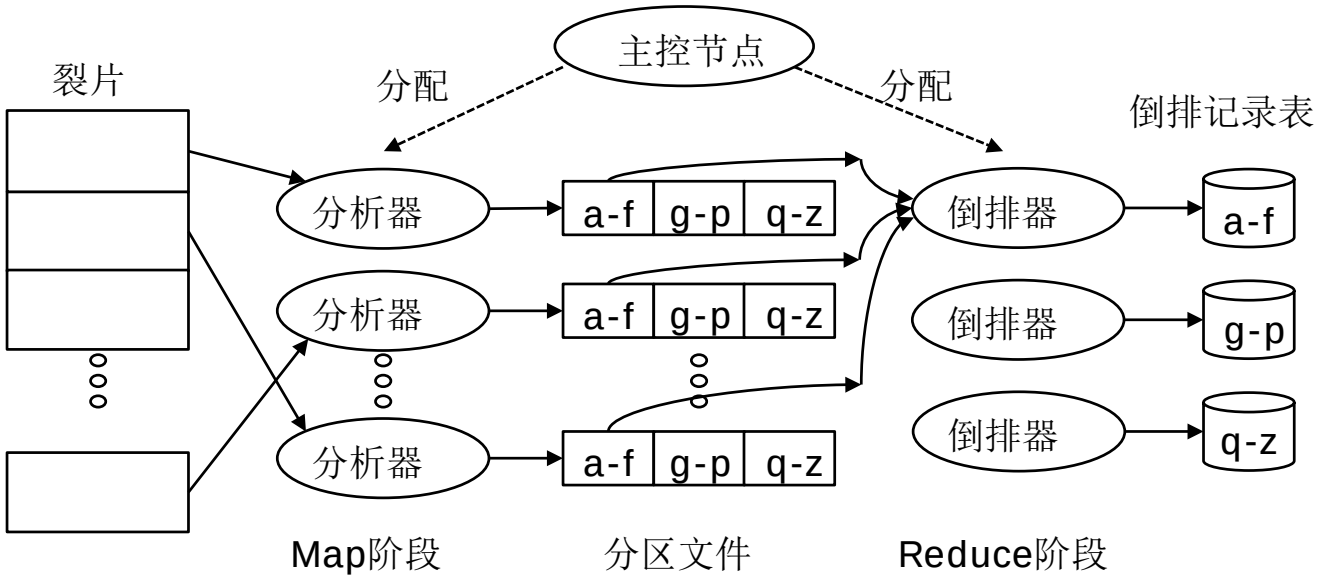
并行任务

- **文档集分割：** 将输入的文档集分片(split) (对应于BSBI/SPIMI算法中的块)
- **两类并行任务**
  - 分析器(Parser)
    - 主节点将一个数据片分配给一台空闲的分析器
    - 分析器逐篇处理文档产生 (term,docID)二元组
    - 分析器将所有二元组分成j 个词项分区，写入j个分区文件
  - 倒排器(Inverter)
    - 主控节点将每一term分区(e.g., a-f分区)分配给一个倒排器，倒排器收集其所有的 (term,docID) 二元组
    - 对每个分区的所有二元组排序，输出倒排记录表（分布式存放）

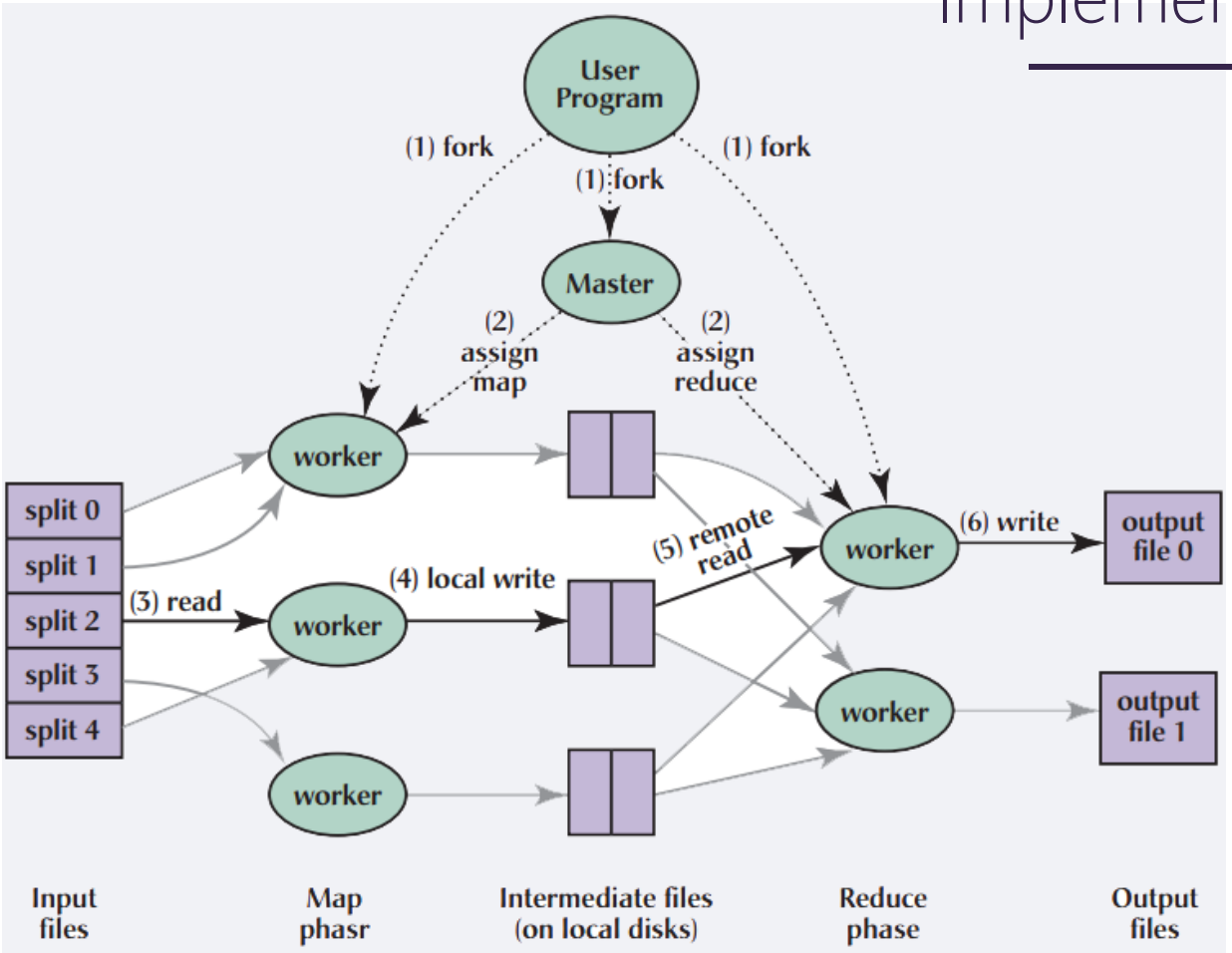
MapReduce

- MapReduce是一个鲁棒的简单分布式计算框架，由Google提出和推广
- Hadoop是MapReduce的一个实现
- 索引构建过程是MapReduce的一个应用实例
- 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
- 需要根据具体问题编写Map和Reduce函数

基于MapReduce索引构建数据流



Implementation-Overview



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(词项ID, 文档ID)
Reduce: (〈词项ID$_1$, list(docID)〉, 〈词项ID$_2$, list(docID)〉, …)	→ (倒排记录表1, 倒排记录表2,…)

索引构建的一个例子

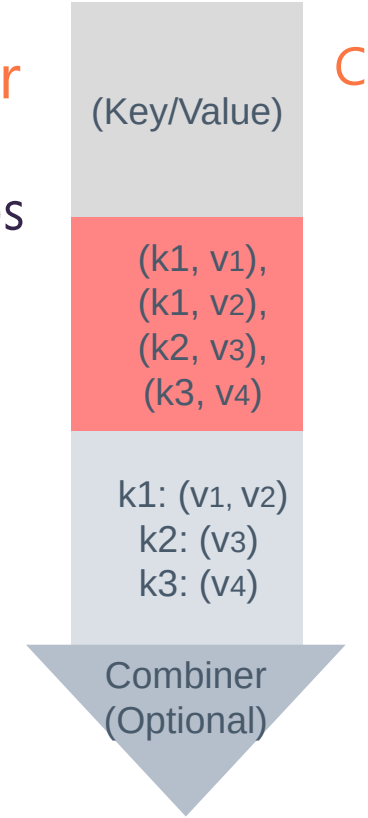
Map: d_2: C died. d_1: C came, C c'ed.	→ (〈C,d_2〉 〈died,d_2〉,〈C,d_1〉 〈came,d_1〉 〈C,d_1〉,〈c'ed,d_1〉)
Reduce: (〈C,(d_2,d_1,d_1)〉,〈died, (d_2)〉,〈came, (d_1)〉,〈c'ed,(d_1)〉)	→ (〈C,(d_1:2,d_2:1)〉,〈died, (d_2:1)〉,〈came,(d_1:1)〉, 〈c'ed,(d_1:1)〉)

（k，list(v)）　⟹　排好序的倒排记录表

Programming Model

- **Map: Written by user**
  - Input: (Key, Value)
  - Intermediate : (k, v)s
  - Output: {k: {v}}

- **Reduce: Written by user**
  - Input: {k: {v}}
  - Output: {k, v}



Counting word occurrences:

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

动态索引构建

- **背景：**文档集是变化的，文档会增加、删除和修改，因此索引应该是动态更新的
- **一种解决方案：**周期性重构一个全新的索引
  - 能够接受对新文档检索的一定延迟
  - 具有足够的资源
- **动态索引构建：**能够及时更新变化文档集索引的方法

动态索引构建：主辅索引法

- 在磁盘上维护一个大的主索引(Main index)
- 在内存中构建新文档的索引，称辅助索引（Auxiliary index)
- 查询处理时，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
- 删除文档的处理：
  - 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

- 合并可能过于频繁（因内存有限）
- 合并时如果正好在搜索，那么搜索的性能将很低
- 合并耗时分析:
  - 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，造成文件系统效率不高
- 另一种方法是将索引整体存入一个大文件中
- 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一并存放在一个文件当中……)

对数合并(Logarithmic merge)

- **目标：** 缓解(随时间增长)索引合并的开销，使用户并不感觉到响应时间上有明显延迟
- **索引组成：** 包含一系列索引，其中每个索引是前一个索引的两倍大
  - 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
- **索引构建过程：** 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, Z0, token)
 1  Z0 ← Merge(Z0, {token})
 2  if |Z0| = n
 3    then for i ← 0 to ∞
 4         do if Ii ∈ indexes
 5              then Zi+1 ← Merge(Ii, Zi)
 6                   (Zi+1 is a temporary index on disk.)
 7                   indexes ← indexes − {Ii}
 8              else Ii ← Zi      (Zi becomes the permanent index Ii.)
 9                   indexes ← indexes ∪ {Ii}
10                   Break
11         Z0 ← ∅

LogarithmicMerge()
1  Z0 ← ∅      (Z0 is the in-memory index.)
2  indexes ← ∅
3  while true
4  do LMergeAddToken(indexes, Z0, getNextToken())
```

对数合并的复杂度

- **索引数目的上界：** $O(\log T)$ (T 是所有倒排记录的个数)
- **对数合并索引构建时间：** $O(T \log T)$
  - 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
- 因此，对数合并的复杂度比辅助索引方式要低一个数量级
- 查询处理时需要合并$O(\log T)$个索引，效率比辅助索引方式低

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

课后作业

- 见课程网页：

  http://10.76.3.31